ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

Scuola di Ingegneria e Architettura
Corso di Laurea Magistrale in Ingegneria e Scienze Informatiche

BLOCKCHAIN: MODELLO GENERALE E TASSONOMIA DELLE COMPONENTI CHIAVE

Relazione finale in
SISTEMI DISTRIBUITI

Relatore
Chia.mo Prof. ANDREA OMICINI

Co-relatori
Dott. GIOVANNI CIATTO

Presentata da
ALEX COLLINI

Terza Sessione di Laurea
Anno Accademico 2016 – 2017

PAROLE CHIAVE

Blockchain

Byzantine Fault Tollerance

Smart contracts

Decentralized systems

Cryptocurrency

A boy asked his Bitcoin-investing
dad for $10.00 worth of Bitcoin currency.
Dad: $9.67? What do you need $10.32 for?

Abstract

Recentemente, l'attenzione dei media e di molte industrie e compagnie si è rivolta a una particolare nuova tecnologia: la blockchain. Questa tecnologia è diventata famosa grazie alla sua applicazione più comune, ossia viene utilizzata principalmente come base per la creazione e gestione di criptomonete. Anche molti "privati" sono entrati nel mondo delle criptomonete, per lo più spinti dal guadagno offerto dall'ormai famoso "mining". Ma cosa si intende per "mining"? Ma un quesito più importante è: "Questa tecnologia è unicamente applicabile al mondo delle criptomonete, oppure è possibile applicarla in diversi campi?". La risposta è: "Sì, è possibile applicarla in diversi campi".

Questa tesi vuole fare ordine per quanto riguarda l'ambiente delle blockchain e separare il concetto di blockchain da quello di "implementazione", andando a creare un modello generale nel quale collocare gli elementi base che compongono ogni qualsivoglia blockchain. Questo perché ogni implementazione ha sì caratteristiche proprie, ma condivide con le altre implementazione una serie di componenti comuni. Viene inoltre analizzato il funzionamento di base che regola questi sistemi. Un altro importante elemento introdotto da alcune implementazioni di questo sistema è il cosiddetto smart contract, la cui analisi dal punto di vista computazionale trova ampiamente spazio in questa tesi. Questi sistemi, inoltre, non sono esenti da problemi dal punto di vista della sicurezza e, essendo sistemi distribuiti, soprattutto per quanto riguarda la comunicazione e la coordinazione tra entità che fanno parte del sistema. Le soluzioni a questi problemi incontrati dalle blockchain vengono analizzate in dettaglio in questa tesi.

Indice

Introduzione

Negli ultimi anni l'interesse intorno alla tecnologia nota come "**blockchain**" sta aumentando esponenzialmente. Con il crescere dell'attenzione mediatica, soprattutto verso **Bitcoin**, la blockchain più famosa a livello mondiale, molte persone e soprattutto aziende si stanno affacciando a questo mondo, molti esercenti stanno iniziando ad accettare pagamenti in Bitcoin e sempre più organizzazioni si stanno appassionando al *mining*.
Purtroppo questa attenzione mediatica, soprattutto per quanto riguarda la facciata economica del sistema, fa sì che diventi credenza comune che una blockchain rappresenti un modo di guadagnare soldi, investendo o facendo "*mining*". Per questo alla blockchain viene assegnato un significato diverso da quello che effettivamente è e le sue funzionalità e aspetti chiave sono sconosciuti ai più, che si fermano al mero aspetto economico. Ma se si va oltre questo aspetto, si può notare come la blockchain riesca a offrire funzionalità molto interessanti per quanto riguarda questo periodo storico, soprattutto il suo possibile impiego nell'ambiente dell'**Internet Of Things** [1].

Come già detto in precedenza, la parola *blockchain* viene sovrastata da altre parole che sono più appetibili per il pubblico, quali "**criptovaluta**", ossia una valuta completamente digitale che viene scambiata attraverso la rete Internet come ad esempio il Bitcoin, "**mining**", ossia la procedura che molti credono "faccia fare soldi facili", ma questa credenza viene screditata solo in parte dall'analisi approfondita presente nei capitoli successivi di questa tesi.
Recentemente, una nuova parola è entrata a far parte dell'ambiente delle blockchain e ha subito suscitato un grande interesse per le funzionalità che sembra offrire: la parola in questione è "**smart contract**". Gli smart contract danno

la possibilità di rendere la blockchain ancora più intelligente e di gestire autonomamente diversi fattori chiave che riguardano la blockchain stessa.

Ad oggi, le più famose e riuscite applicazioni della blockchain sono principalmente due: **Bitcoin** e la nuova rivelazione che porta con sé una ventata di cambiamenti, **Ethereum**. Entrambe sono implementazioni di **sistemi distribuiti** basati su blockchain che hanno acquisito importanza e fama grazie alla loro criptovaluta nativa, il *Bitcoin* e l'*Ether* rispettivamente, ma un sistema basato su blockchain può essere molto versatile. Ad esempio è stata effettuata una proposta di creare una nuova rete Internet completamente decentralizzata, sfruttando le caratteristiche proprie della blockchain [2]. Tuttavia la maggior parte delle blockchain che si stanno sviluppando e diffondendo tengono a mente solo l'aspetto economico della faccenda, creando criptovalute su criptovalute, non sfuttando appieno il vero potenziale della blockchain.

Questa tesi si pone l'obiettivo di creare dell'ordine per quanto riguarda l'argomento **blockchain**: cos'è effettivamente una blockchain? Da cosa è composta e quali sono i princìpi che la regolano? Un altro importante obiettivo consiste nel separare il concetto di blockchain da quello di "implementazione di una blockchain", creando un modello generale nel quale trovino collocazione gli elementi base che compongono ogni tecnologia che voglia definirsi "blockchain". Inoltre, si va ad analizzare il micro livello delle varie blockchain, specificando quali meccanismi siano imprescindibili e quali possano essere considerati come aggiunte al sistema, per migliorare performance o sicurezza. Come già anticipato, esistono diverse tecnologie che utilizzano un sistema basato su blockchain. Per questo, viene poi effettuata una panoramica di alcune implementazioni, con annesso confronto. Si va poi ad approfondire il concetto di **smart contract** a cui si è accennato precedentemente, cercando di dare una definizione generale e di spiegare la semantica che li regola.

La tesi è strutturata in questo modo: nel Capitolo 1 viene introdotto il modello generale di una blockchain, spiegandone i concetti base, opportunamente epurati da qualsiasi dettaglio implementativo, introducendone inoltre la semantica

base. Nel Capitolo 2 si parla delle problematiche che le blockchain, in quanto sistemi distribuiti, devono affrontare durante la loro vita e di come le varie implementazioni siano riuscite a gestire tali problemi. Nel Capitolo 3 vengono definiti gli smart contract ed è presente inoltre un'analisi per quanto riguarda l'aspetto computazionale e gli aspetti critici che li riguardano. Nel Capitolo 4 vengono analizzate e confrontate tra di loro alcune implementazioni interessanti, alla luce di quanto detto nei capitoli precedenti, per giungere poi alle conclusioni, nel Capitolo 5, dove sono presenti delle speculazioni per quanto riguarda il futuro della tecnologia.

Capitolo 1

Modello generale

In questo capitolo vengono presentati i principali elementi che modellano una tecnologia basata sulla blockchain, insieme alla semantica che regola il comportamento della blockchain stessa.

1.1 Definizioni

Tutti i sistemi basati su **blockchain** condividono una serie di elementi fondamentali, analizzati in questa sezione, che permettono loro di comportarsi in maniera coesa e coerente.

Nonostante ci siano numerose implementazioni di questo tipo di tecnologia, ognuna con caratteristiche proprie, è possibile isolare alcune caratteristiche comuni. Di seguito vengono elencate queste caratteristiche, ognuna con una propria denominazione che verrò utilizzata all'interno di questo capitolo. Un sistema basato su blockchain è composto dai seguenti elementi:

- un'insieme di **blocchi** (B);
- un'insieme di **stati** (S);
- un'insieme di **identificatori**, o **indirizzi** (I);
- un'insieme di **transazioni** (TX);

- una **funzione** che verifica la **validità** di una **transazione** (F_{tx});
- una **funzione** che **verifica** la **consistenza** di uno **stato** (F_s);
- una **funzione** che effettua l'**ordinamento** delle **transazioni** (F_o);
- un blocco iniziale, o **genesis block** ($b_0 \in B$);

Di seguito, vengono descritti in maniera generale gli elementi sopracitati, senza scendere nel dettaglio implementativo, con lo scopo di creare una tassonomia dei sistemi blockchain.

1.1.1 Stato

Per rendere più immediata la comprensione del concetto di stato, si procede alla sua descrizione tramite un semplice esempio. Alice e Bob si affidano a un sistema basato su blockchain che gestisce una criptovaluta nativa. I due decidono di effettuare una prima "operazione" che va ad assegnare loro un "valore" iniziale, che può essere rappresentato da qualsiasi tipo di dato X. La loro situazione iniziale può essere raffigurata così:

$$A_0 : 20\ X$$
$$B_0 : 30\ X$$

A_0 e B_0 si dicono **stati** (s_i) di una blockchain e identificano la situazione attuale di un determinato elemento della blockchain. È stata usata la parola "elemento" perché, come vedremo nei capitoli successivi, ogni blockchain ha la propria rappresentazione e implementazione di stato, che può differire o essere simile a quella di altre blockchain. Infatti, i valori inseriti nell'esempio riportano in maniera generale una rappresentazione simile a quella che è possibile trovare all'interno di una blockchain che gestisce una criptovaluta nativa. Nel Capitolo 2 vengono analizzate più nel dettaglio queste rappresentazioni. Il passaggio da uno stato a un altro viene chiamato **transazione**.

1.1.2 Transazione

Ad un certo punto, Alice, tramite un acquisto online, deve trasferire del denaro a Bob. Per comodità d'esempio, non vengono considerati i vari vincoli che possono essere imposti a una transazione del genere, quali verifica della disponibilità del saldo dell'account dell'acquirente e altri ancora. Si ha quindi un cambiamento di stato, una **transazione** (tx_i), che fa sì che i soldi necessari vengano sottratti dallo stato precedente del conto di Alice e vengano poi depositati sul conto di Bob. Si avrà una situazione del genere:

$$tx_i = \begin{cases} A_0 \overset{-10}{\rightarrow} A_1 \\ B_0 \overset{+10}{\rightarrow} B_1 \end{cases}$$

che produrrà un effetto del genere:

$$A_1 : 10\ X$$
$$B_1 : 40\ X$$

La transazione viene successivamente inserita all'interno di un contenitore, dove sono presenti tutte le transazioni che sono state effettuate in un determinato momento all'interno della blockchain. Questo elemento prende il nome di **blocco**.

Il concetto di transazione è generale e non deve essere assolutamente scambiato per un qualcosa strettamente legato alla mondo finanza (ad esempio, può considerarsi una transazione il passaggio di proprietà di un auto da un proprietario A a un proprietario B, ma anche il semplice invio di dati da parte di un sensore che genera una transazione sulla blockchain e quindi un cambiamento di stato).

1.1.3 Blocco

Come detto in precedenza, il **blocco** (b_i) è una struttura dati al cui interno viene inserita una lista di transazioni. Questi blocchi hanno una struttura che varia in base all'implementazione scelta dalla blockchain, ma ci sono alcuni

elementi comuni, come l'*hash* del blocco precedente, il *timestamp* e il *nounce*. Questi elementi vengono analizzati in dettaglio nel Capitolo 2.
Naturalmente, non esiste un blocco unico, ma una serie di blocchi che sono uniti, "incatenati" (da qui il nome *blockchain*) tra di loro tramite un meccanismo noto come *hash chain*, analizzato nei capitoli successivi di questa tesi. Questo insieme di blocchi va a formare un *ledger* (libro mastro) che parte dal primo blocco in assoluto (il cosiddetto *genesis block*), fino ad arrivare al blocco più recente. Così facendo si viene a creare una cronistoria di tutte le transazioni che sono avvenute all'interno del sistema dal momento della sua creazione ed è questa l'essenza della blockchain, un libro mastro distribuito decentralizzato o, secondo chiave di lettura informatica, un database decentralizzato.
Trattandosi di un sistema decentralizzato, vi è la completa assenza di qualsivoglia entità centrale che controlli o gestisca l'intero sistema. Si tratta, invece, di una rete peer-to-peer formata da **nodi** che contengono al loro interno una copia della blockchain, che viene aggiornata volta per volta all'inserimento di nuovi blocchi o in seguito a eventi di varia natura, come *fork*.

1.1.4 Nodo

La vera potenza di un sistema basato su blockchain è la versatilità del sistema stesso, le cui fondamenta poggiano su una rete peer-to-peer composta da entità che svolgono determinate funzioni all'interno della rete; queste entità vengono chiamate **nodi** (n).
I nodi non hanno tutti lo stesso ruolo all'interno di una rete, ma alcuni ricoprono funzioni particolari, che possiamo quasi definire come *servizi*. Naturalmente, ogni implementazione ha i propri nodi "speciali" che vanno a rispecchiare le necessità e le scelte implementative del sistema, ma ci sono alcuni ruoli che possiamo definire comuni tra le implementazioni, ma solo perché incarnano funzioni intrinseche al concetto di blockchain e di sistema decentralizzato: possiamo trovare nodi che generano transazioni, nodi che generano blocchi, nodi che validano blocchi, nodi che regolano le identità degli utenti rilasciando certificati firmati e così via, alcuni dei quali verranno presentati nei capitoli successivi. Ci sono poi dei nodi particolari, i *miners*, che aiutano a mantenere

il sistema efficiente e reattivo, pur non mantenendo una copia integrale della blockchain.

1.2 Descrizione del funzionamento

La semantica operazionale che si sta per descrivere è comune a tutte le implementazioni dei sistemi basati su blockchain.

Quando si istanzia per la prima volta un sistema viene creato il blocco iniziale b_0, o **genesis block**. Questo blocco deve necessariamente essere considerato come valido da qualsiasi nodo n partecipante al sistema. Il *genesis block* rappresenta lo stato iniziale del sistema. A partire da questo, il sistema inizia il "ciclo di vita" vero e proprio.
La prima fase è la pubblicazione di un nuovo blocco che avverrà in un certo momento t. In particolare, indicando con t_0 l'istante di pubblicazione del blocco b_0, ossia il momento in cui il sistema ha avuto origine, si sa che il tempo di pubblicazione t_{i+1} del blocco b_{i+1} equivarrà a $t_i + \Delta t$, dove Δt indica il periodo di tempo impiegato dai nodi a pubblicare un nuovo blocco e t_i indica l'istante di pubblicazione del blocco precedente. Δt non è da considerare come un valore totalmente casuale, ma come una variabile aleatoria che segue una funzione matematica ben precisa, ossia la funzione di densità di probabilità [3]. Ogni blocco b_i contiene al suo interno una lista di transazioni tx_i che sono state effettuate dagli utenti che partecipano al sistema in un periodo di tempo che va dal momento della creazione del sistema (t_0) fino al momento della pubblicazione del blocco (t_i) non compreso, ordinate da una certa funzione di ordinamento F_o. La funzione F_o permette alla rete di stabilire l'ordine in cui sono avvenute le transazioni e ogni nodo n deve necessariamente rispettare questo ordine. Queste transazioni sono uniche rispetto al blocco corrente e a quelli precedenti, ciò vuol dire che nel blocco $b_i + 1$ non saranno presenti le stesse transazioni presenti all'interno del blocco b_i. Questi blocchi, prima di essere pubblicati, e di conseguenza anche le transazioni al loro interno, devono essere considerati sia **validi** che **consistenti**. Una transazione viene considerata valida se e solo se la funzione che verifica la validità delle transazioni (F_{tx})

la considera tale. Di conseguenza, un blocco b_i viene considerato tale se ogni transazione tx_i al suo interno è valida. Per quanto riguarda la consistenza, una transazione t_x si definisce consistente se e solo se lo stato risultante da quella transazione risulta essere consistente. Uno nuovo stato s_{i+1} viene computato a partire da uno stato s_i data una determinata transazione tx_i. Lo stato s_{i+1} viene considerato consistente se e solo se viene considerato tale dalla funzione che verifica la consistenza degli stati F_s. Anche in questo caso, un blocco risulta consistente se tutte le transazioni al suo interno sono consistenti.
Una volta che il blocco b_i viene considerato valido e consistente, il sistema procederà alla computazione del nuovo stato s_i eseguendo ricorsivamente tutte le transazioni tx_i contenute del blocco b_i allo stato s_{i-1}.

Con questa operazione si conclude uno step del "ciclo di vita" di un sistema basato su tecnologia blockchain. Questo step verrà poi ripetuto ogni qual volta i nodi pubblicheranno un nuovo blocco.
Il momento in cui viene avviata l'esecuzione dei noti **smart contract**, che vengono ampiamente analizzati nel Capitolo 3, si colloca all'interno dell'esecuzione delle transazioni.

Questo capitolo generalizza il più possibile le componenti base e la semantica che regola il comportamento e l'interazione tra le entità che prendono parte a questo tipo di sistema e per dare una base teorica utile alla comprensione dei capitoli successivi di questa tesi. Questo perché ogni variante di questa tecnologia ha caratteristiche proprie per quanto riguarda le scelte implementative, ma ognuna di queste segue le regole dettate dalla semantica sopra descritta.

Capitolo 2

Problematiche nel mondo reale

In questo capitolo verranno introdotte alcune problematiche proprie dei sistemi distribuiti e che quindi sono state ereditate dalle tecnologie blockchain in quanto sistemi distribuiti. Come primo problema verrà analizzata la concorrenza e il recupero delle informazioni, con l'introduzione del concetto di *asset*. Successivamente si parlerà di ordinamento degli eventi, un problema legato alla mancanza di clock globale nei sistemi distribuiti. Si passerà poi a parlare dell'identità degli utenti in un sistema basato su tecnologia blockchain, ossia come ogni sistema gestisce e garantisce le identità degli utenti che prendono parte al sistema. Successivamente, l'attenzione si sposterà al problema di rendere inviolabile il passato, ossia si discuterà degli stratagemmi che le blockchain hanno adottato per prevenire che un utente malintenzionato possa cambiare a piacimento ciò che è già stato memorizzato nel sistema. Un altro punto che verrà analizzato riguarda gli attacchi che possono essere sferrati ai sistemi basati su blockchain. Infine, si parlerà di incentivi e disincentivi economici usati dalle tecnologie blockchain per far sì che i propri utenti seguano le regole.

2.1 Concorrenza e recupero delle informazioni

Uno dei principali problemi che ogni sistema informatico deve affrontare durante la sua vita è la concorrenza. Un sistema ha un buon fattore di concorrenza se permette a diversi processi di funzionare simultaneamente e in un

qualsiasi ordine, riducendo ritardi e quindi andando a generare output maggiori rispetto a un sistema non concorrente. Un sistema può gestire simultaneamente un insieme di operazioni soltanto se queste operazioni sono indipendenti tra di loro, altrimenti, se esistesse un qualche legame tra operazioni, l'ordine in cui verrebbero processate potrebbe dare risultati diversi. Per capire al meglio come la concorrenza viene gestita all'interno delle blockchain dobbiamo entrare nel dettaglio di come le transazioni vengono gestite all'interno della blockchain stessa.

Introduciamo ora un concetto che ci tornerà utile in questa sezione, ovvero cos'è un **asset**. Un asset è un qualsiasi oggetto di proprietà di un soggetto che può avere un valore economico. Questa definizione [4] è molto lasca, perché qualunque cosa può assumere un valore economico se considerata "merce di scambio". Le blockchain possono essere impiegate in più casi d'uso [1], come registri per il tracciamento di asset di diverso tipo, come spedizioni, auto, insieme a numerosi altri beni che possono essere considerati come asset. La domanda che sorge spontanea è: "Come fare a rendere digitale una cosa che non può essere resa digitale?" Ad esempio, acquistare un libro da uno store online è possibile grazie alla "digitalizzazione" del denaro presente sul nostro conto corrente, ma per i beni materiali o immateriali diversi dal denaro si è deciso di adottare una soluzione che porta alla digitalizzazione del bene: ossia renderlo un gettone, o **token**. Ogni token viene collegato a un determinato bene e viene assegnato a un proprietario. Questo proprietario può tenerlo per sé, dimostrando quindi di avere possesso di quel determinato bene, oppure cederlo a un altra persona, la quale diventerà il nuovo proprietario del bene. Questo tipo di asset prende il nome di **tokenized asset**.

Ora che abbiamo introdotto il concetto di asset, procediamo con la spiegazione di come questi asset possono essere scambiati all'interno delle blockchain. Immaginiamo la blockchain come un libro mastro che al suo interno ha diverse righe che contengono informazioni sulle varie transazioni avvenute nel sistema. Ad esempio, se Alice possiede 10 unità di un *asset* X, nel libro mastro deve essere presente una riga che corrisponda a questo stato: quindi avremo

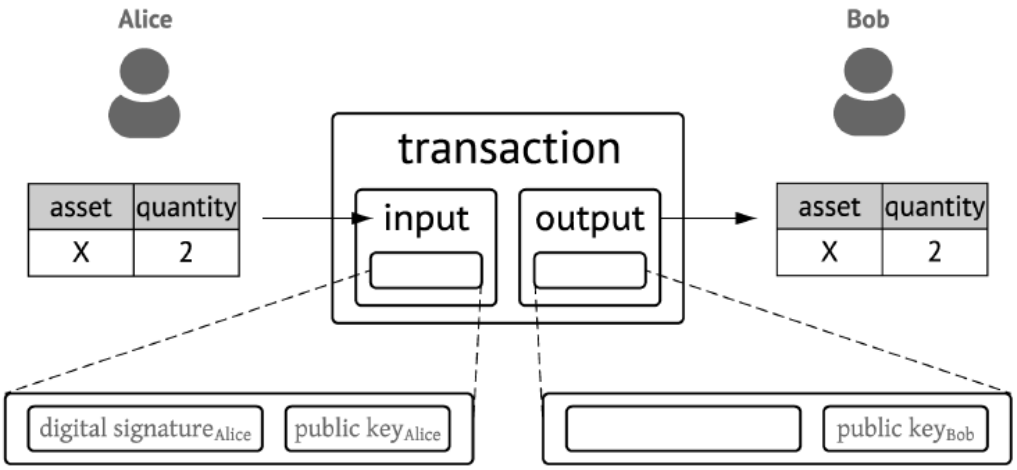

Figura 2.1: Transazione che trasferisce un parte di un asset, o tokenized asset, da Alice a Bob. Possiamo notare come Alice abbia firmato la sua transazione in input e come abbia bloccato la transazione sulla chiave pubblica di Bob, facendo si che solo Bob possa spenderla. [1]

diverse colonne che conterranno le informazioni che rappresentano lo stato, come "proprietario", "tipo di asset" e "quantità", con l'unico appunto che nel campo "proprietario" non troveremo la stringa "Alice", ma un identificativo che rispecchi l'identità di Alice, ad esempio la sua chiave pubblica (questo argomento verrà trattato più in dettaglio nelle sezioni successive). Supponiamo che Alice voglia inviare 2 unità del suo asset X a Bob: Alice non deve fare nient'altro che firmare una transazione che andrà a creare una nuova riga nel database dove verrà decretato il passaggio di 2 unità da Alice a Bob, inserendo la chiave pubblica di Bob come nuovo proprietario di quelle 2 unità. La Figura 2.1 illustra in maniera semplificata questa operazione. Per quanto riguarda la riga dove veniva specificato il possesso delle 10 unità da parte di Alice, essa viene sostituita da un nuova che rispecchia l'attuale bilancio di Alice (8 unità). Tutte le righe che non sono state ancora cancellate, perché non sono state ancora consumate da altre transazioni, vengono chiamate *unspent transaction outputs* (**UTXO**). Con questa espressione ci si riferisce anche al modello che viene usato da Bitcoin e da tutte quelle blockchain *Bitcoin-like*, cioè che prendono spunto dall'implementazione di Bitcoin.

A opporsi alla filosofia **UTXO**, abbiamo il modello **account-based**, che viene utilizzato nella ormai famosa tecnologia Ethereum. Come è facilmente intuibile dal nome del modello, il sistema permette a chi lo utilizza di gestire dei veri e propri account, ognuno con il proprio indirizzo assegnatoli dal sistema. Ethereum, come tutte le blockchain che utilizzano un modello *account-based*, ha uno stato globale che può essere immaginato come la lista degli account del sistema, con tutte le informazioni relative all'account, come l'indirizzo e il bilancio.

Scegliere l'uno o l'altro modello sta a come si vuole implementare la propria blockchain, non c'è un modello consigliato oppure migliore dell'altro, ma naturalmente ogni modello ha i propri vantaggi e svantaggi in relazione alle applicazioni. Alice, ad esempio, vuole vedere a quanto ammontano le sue finanze in una data blockchain. Decide di accedere al suo *wallet*, cioè un'applicazione client che segue le regole implementative della blockchain di riferimento, e da lì riesce a vedere quanti soldi ha in quella determinata blockchain. Ma la vera domanda è: "è effettivamente così semplice anche "sotto il cofano" dell'applicazione?". La risposta è, come ci si può aspettare, "dipende dal modello scelto dalla blockchain".
Infatti, in una blockchain che adotta il modello *UTXO*, l'operazione di ricostruire l'ammontare dei fondi di un utente non è poi così banale per quanto riguarda l'aspetto implementativo. Un *wallet* di una blockchain con modello *UTXO* deve ricercare in tutta la blockchain qualsiasi transazione non spesa, e quindi ancora valida, di un determinato account e poi successivamente sommare il valore di tutte le transazioni per ottenere il saldo dell'account. Un problema implementativo ancora più ostico è rappresentato dalla funzione che genera una transazione *send*, quindi un trasferimento di un asset da un indirizzo all'altro. Questa funzione quindi deve raccogliere tutte le *UTXO* dell'account, poi da quell'insieme di *UTXO* deve estrarre un sottoinsieme dal valore complessivo uguale o maggiore del valore degli input che si vogliono utilizzare per quella transazione. In un modello *account-based* la questione è molto più semplice, perché è tutto memorizzato all'interno di un qualcosa pa-

ragonabile a come una banca gestisce i conti dei propri clienti, ma in maniera totalmente decentralizzata. Per far sì che una transazione vada a buon fine basta controllare che il saldo dell'account sia sufficiente per pagare il costo della transazione.
Un altro confronto interessante tra *UTXO* e *account-based model* è sulla gestione degli stati. Le blockchain *UTXO* sono *stateless*, quindi sono transazioni che non ammettono stati, cosa che invece gli *smart contracts* (tecnologia che viene usata dalle blockchain *account-based*) ammettono, quindi una blockchain basata su *UTXO* può solamente essere utilizzata per trasferimenti immediati e che non richiedono una temporizzazione o la memorizzazione dello stato della transazione. Ad esempio, in un ambiente *account-based*, quindi tramite utilizzo di *smart contracts* invocabili tramite indirizzo statico assegnatoli dalla blockchain al momento del deploy, è possibile creare un contratto che gestisca in maniera del tutto autonoma, tramite del codice presente all'interno del contratto, delle transazioni che possono cambiare lo stato degli oggetti presenti sulla blockchain.

Riassumendo, nel modello **UTXO**:

- ogni transazione inserita in input deve essere valida e non spesa;
- la firma di ogni transazione inserita in input deve corrispondere a quella del proprietario;
- il totale del valore delle transazioni in input deve essere uguale o maggiore al totale del valore della transazione in output.

Per quanto riguarda il modello **account-based**:

- esiste uno stato globale con una lista di account presenti nel sistema;
- una transazione viene considerata valida se l'account che ha avviato la transazione dispone di fondi a sufficienza per pagare il costo della transazione ed è necessaria solo la firma del mittente;
- è possibile scrivere codice all'interno di contratti per automatizzare alcune procedure.

2.2 Ordinamento degli eventi

Riprendendo quanto detto nella sezione precedente, quando si parla di concorrenza si parla anche di informazioni che possono giungere a un nodo in maniera diversa rispetto a un altro nodo. Pertanto è necessario che tutto il sistema sappia esattamente in che modo e ordine debba gestire le informazioni che giungono ai diversi nodi, per evitare problemi interni al sistema e, a volte, lo sfaldarsi dello stesso. Facciamo un esempio per capire perché l'ordine delle transazione è importante.

Come sempre, l'ambiente finanziario permette di fare esempi che possono far capire con immediatezza il problema. Alice e Bob questa volta hanno un conto in comune che ammonta a 1000$. I due si trovano in città diverse, Bob ha bisogno di 900$, mentre Alice ha bisogno di 200$ e provano **contemporaneamente** a ritirare i propri soldi dal conto in comune. Insieme, le due transazioni **non** possono andare a buon fine, perché i due stanno ritirando più soldi di quelli che sono presenti nel loro conto, ma solo una delle due può andare a buon fine.
Però c'è un problema: quale delle due operazioni va a buon fine? Verrebbe naturale da dire "quella che viene ricevuta per prima", ma il "per prima" è un concetto molto critico quando si parla di sistemi distribuiti, soprattutto perché entrano in gioco le reti di comunicazione e i numerosi fattori che devono essere presi in considerazione quando si parla di reti, sia per quanto riguarda un sistema centralizzato, sia per quanto riguarda un sistema decentralizzato, come ad esempio la mancanza di **clock globale** nei sistemi distribuiti. Il clock globale è un meccanismo di coordinazione che detta il tempo in un sistema distribuito. Tanenbaum e Steen [5] fanno un esempio molto semplice quanto efficace che spiega la necessità di un meccanismo che dia un ordine su cui tutti i partecipanti al sistema devono basarsi. Si prenda la funzione *make*, ossia una funzione dei sistemi operativi *UNIX* che permette di ricompilare i file sorgenti di un programma a seguito di modifiche effettuate dal programmatore. *Make* esamina quando è stata effettuata l'ultima modifica di un file: se il file *input.c*, ossia il file sorgente, ha, ad esempio, 2151 come **timestamp**, ovvero una qualunque

marca temporale che identifichi il momento esatto in cui è stata apportata una modifica a quel file, e il file compilato, *input.o*, ha 2150 come timestamp, *make* rileverà che *input.c* è stato modificato e che quindi deve essere ricomplilato. Si immagini ora cosa potrebbe accadere se questa situazione capitasse in un sistema distribuito senza una sorta di accordo sul clock globale. Si supponga che il file compilato *input.o* abbia 2144 come timestamp, poco dopo il file sorgente *input.c* viene modificato da un utente e gli viene assegnato 2143 come timestamp, perché la macchina dell'utente aveva un clock leggermente inferiore. Così facendo, *make* non ricompierà quel sogente, visto che il timestamp del file compilato è più recente rispetto a quello del file sorgente. Quindi, la mancanza di un clock globale può essere causa di problemi di comunicazione tra entità che partecipano a un sistema distribuito.

Inoltre, è da tenere in considerazione anche il tempo di propagazione delle informazioni sulla rete, in quanto le informazioni, in questo caso le transazioni di Alice e Bob, vengono immesse in rete devono viaggiare attraverso numerosi nodi prima di arrivare a destinazione ed essere elaborate. C'è quindi bisogno di un algoritmo distribuito che decida l'ordine in cui le transazioni devono essere prese in considerazione, in modo da accordare l'intero sistema su un'unica versione dei fatti.

Nell'ambiente della blockchain si parla di **consenso**, una parola che, come significato, spiega in maniera esaustiva qual è il suo scopo finale: creare *consenso* tra i vari nodi che partecipano alla rete, andando quindi a creare un ordine in un ambiente completamente *trustless* e distribuito. Il *consenso* non serve semplicemente a ordinare le transazioni, ma è anche una specie di prima barriera contro comportamenti malevoli da parte dei nodi.
Non esiste semplicemente un unico algoritmo di consenso, infatti Cachin e altri [6] hanno classificato gli algoritmi di consenso in tre macrogruppi: **decentralized consensus**, **somewhat decentralized consensus** e **consortium consensus**. Negli algoritmi di consenso decentralizzato (*decentralized consensus*) i nodi preparano i blocchi contenenti una lista di transazioni valide, si passa poi a una sorta di gara (*lottery race*) dove tutti i nodi devono risolvere

un puzzle che prevede un alto uso di risorse computazionali da parte di ogni nodo. Viene poi scelto il nodo vincitore di questa gara in maniera casuale e il suo blocco verrà considerato come prossimo da inserire sulla blockchain e, come premio, riceverà un una ricompensa in termini di criptovaluta; successivamente tutti i nodi verificheranno e valideranno il nuovo blocco che verrà inserito nel branch più lungo della blockchain. Questo tipo di consenso viene sfruttato principalmente da blockchain di tipo *permissionless* (la differenza tra blockchain *permissionless* e *permissioned* verrà spiegata nella sezione successiva), come Bitcoin, dove non è presente nessun tipo di entità centrale che gestisce il *consenso*. Questo approccio richiede una *proof-of-work*, o **POW**, che attesti che il nodo ha risolto il puzzle computazionale per quel blocco e che non ha cercato escamotage per ottenere vantaggi senza risolvere il puzzle.
Il tipo di consenso *somewhat decentralized* mette a disposizione una sorta di organo di controllo centralizzato: ogni nodo sceglie dei vicini a cui è connesso e di cui si fida, ogni transazione tra due nodi è ritenuta valida se viene accettata dalla maggioranza delle entità in comune presenti nel loro set di nodi "di fiducia"; si ha quindi una sorta di centralizzazione.
Si passa poi al *consortium consensus*, usato principalmente da blockchain *permissioned*, dove esiste un set di nodi validatori già prestabiliti (i nodi che fanno parte del gruppo dei nodi validatori possono cambiare durante la vita del sistema), e vengono usati degli algoritmi di consenso molto particolari, che fanno parte della famiglia dei *Byzantine fault tollerance algorithms*, o algoritmi **BFT**. Questo gruppo di algoritmi prende il nome dal problema dei generali bizantini: il problema parla di un gruppo di generali che stanno decidendo il piano d'azione per assaltare una città e tutti i generali devono rispettare la decisione presa collettivamente. Il problema viene complicato dalla presenza di un generale traditore che non solo potrebbe votare per una strategia non ottimale, ma potrebbe anche riferire a una parte del gruppo un ordine opposto a quello che è stato dato dai propri superiori (ad esempio, dire di attaccare quando invece è stato ordinato di ritirarsi). La cosa può essere tranquillamente trasposta all'ambiente della blockchain: infatti dei nodi (*faulty nodes*) possono comportarsi in maniera diversa in base all'"interlocutore" creando quindi discrepanze nel consenso. Un algoritmo BFT tollera f su n *faulty nodes*, infatti è

impossibile rendere un sistema *Byzantine-fault tollerant* se il numero di *faulty nodes* è maggiore o uguale a un terzo dell'intero insieme di nodi.

I due approcci per il consenso (*POW* e *BFT*) possono essere comparati secondo una serie di proprietà intrinseche dei sistemi. Vukolić [7] propone un confronto tra *POW* e *BFT* su vari aspetti della blockchain analizzati in base a quale algoritmo di consenso viene utilizzato. L'analisi di questa tesi si concentra solo sugli aspetti più importanti, quali la *consensus finality*, la *scalabilità* sia per quanto riguarda il numero di nodi che il numero di client e la *performance* in generale.
La *consensus finality* è la proprietà che afferma che un blocco valido, una volta aggiunto alla blockchain, non possa più essere rimosso dalla blockchain. Questa proprietà non viene soddisfatta dalle blockchain con *POW* perché sono inclini a *fork* temporanee: una *fork* è una situazione che si viene a creare quando non si riesce a trovare un consenso unanime su un determinato blocco e, a partire da quel blocco in poi, si verrà a creare un altro ramo della blockchain, parallelo a quello da cui è partito il fork. Prima o poi il fork si risolverà grazie a diverse regole che sono implementate nella blockchain (ad esempio la regola applicata da Bitcoin, ossia viene considerato valido il branch composto dal maggior numero di blocchi e quindi dopo un certo periodo di tempo tornerà a essere il branch principale su cui si baserà la blockchain per le transazioni future). D'altro canto, la *consensus finality* viene rispettata nelle blockchain *BFT*, dove gli utenti hanno l'immediata conferma dell'inclusione della loro transazione all'interno della blockchain.
Per quanto riguarda la **scalabilità** dei nodi, le blockchain *POW* sono molto scalabili, ne è un esempio lampante Bitcoin con i suoi numerosissimi [8] nodi, al contrario delle blockchain *BFT*, le quali hanno una scalabilità limitata (numero di nodi compreso tra 10 e 20); entrambi gli approcci, però, scalano molto bene per quanto riguarda il numero di client collegati alla rete.
Completamente opposto è lo scenario delle *performance*: le due principali sfide per le blockchain *POW* sono la *block size* e la *block frequency*, ossia la dimensione di un blocco e ogni quanto viene creato un blocco. Aumentare la dimensione massima dei blocchi con lo scopo di incrementare il numero di transazioni fa

aumentare di conseguenza la latenza a causa dei maggiori ritardi di propagazione attraverso la rete Internet dovuta alla dimensione dei blocchi, creando possibili aperture a problemi di sicurezza (maggiori ritardi possono facilitare il verificarsi di fork); gli stessi problemi di sicurezza possono presentarsi anche aumentando la frequenza dei blocchi con lo scopo di ridurre la latenza dovuta dal meccanismo di conferma dei blocchi. Al contrario, le blockchain *BFT* riescono a servire numerose transazioni contemporaneamente ed essere limitate solamente dalla latenza della rete.

2.3 Identità degli utenti

In un ambiente *trustless* come quello delle blockchain deve esserci un meccanismo in grado di garantire l'identità degli utenti. Dare un nome *human-readable* a ogni nodo è sia dispendioso in termini di tempo sia poco sicuro, soprattutto perché un nodo malevolo potrebbe rubare l'identità di un altro nodo e spacciarsi per lui. Come fanno i nodi a instaurare della *trust* in un ambiente del genere? La soluzione è utilizzare la **crittografia asimmetrica**. La *crittografia asimmetrica* prevede la creazione di due chiavi per ogni nodo, la **chiave privata** e la **chiave pubblica**: tramite questo metodo si va a garantire **autenticità**, ossia un nodo può essere certo che un determinato messaggio, se crittografato in maniera corretta, è stato effettivamente inviato dal mittente che si aspettava. [9]
Il funzionamento è molto semplice e intuitivo: Alice vuole mandare un messaggio a Bob, allora scrive il messaggio e lo cifra con la sua *chiave privata*, che solo lei conosce e solo lei può avere. Alice invia il messaggio e la sua versione cifrata a Bob. Il messaggio viene ricevuto da Bob che, essendo in un ambiente *trustless*, vuole essere sicuro che il messaggio sia stato effettivamente mandato da Alice. Allora Bob prende la *chiave pubblica* di Alice, reperibile da tutti, e decifra il messaggio con quella chiave: se tutto va a buon fine Bob è sicuro il mittente sia Alice. Un messaggio che viene cifrato con una delle due chiavi generate può essere decifrato solo dall'altra chiave.
Lo schema applicato dalla crittografia asimmetrica è molto utile in questo tipo di sistema perché gode di proprietà che snelliscono alcune operazioni; infatti

per ogni nodo è facile generare una coppia di chiavi e decifrare un messaggio con la chiave corretta, mentre garantisce una prima barriera per gli attaccanti, perché risulta davvero difficile, se non impossibile, decifrare un messaggio con la chiave sbagliata oppure riuscire a trovare una chiave a partire dall'altra.

Inoltre, la chiave pubblica può fungere da identificatore dell'utente che partecipa a una determinata blockchain. Bitcoin ne è un esempio e molte altre blockchain pubbliche, dove la barriera d'accesso è minima e si può entrare subito a far parte della blockchain senza troppi problemi. Queste blockchain pubbliche dove tutti possono partecipare vengono chiamate **permissionless** e non c'è un rigido controllo per quanto riguarda le identità dei nodi. I nodi non conoscono l'intero insieme dei peer che fanno parte del sistema, ma un sottoinsieme contenente solamente una porzione dei peer vicini.
A opporsi alle blockchain *permissionless* troviamo le blockchain private o **permissioned**: questo tipo di blockchain mette a disposizione un ente, che possiamo definire centrale, la cosiddetta **Certification Authority** che gestisce le identità dei vari nodi, rilasciando anche certificati digitali che attestano le identità dei nodi.

Come è facilmente intuibile, i due modelli di blockchain non sono applicabili a ogni scenario in maniera indifferente: gli ambienti aziendali prediligono un sistema *permissioned*, dove è necessario sapere l'identità di ogni nodo che prende parte al sistema, cosa che una blockchain *permissionless* non può garantire; inoltre nelle blockchain *permissioned* solo un gruppo ristretto di nodi può partecipare al consenso, al contrario delle blockchain *permissioned* dove tutti possono partecipare alla validazione dei blocchi per creare consenso. Un'altra differenza importante tra blockchain *permissionless* e *permissioned* è il modello di *mining* sottostante alla blockchain: le blockchain *permissionless* sfruttano la *proof-of-work*, quindi i *miner* mettono a disposizione la propria potenza computazionale, o *hashing power*, al sistema. Finché il 51% dei partecipanti al sistema seguono le "regole del gioco" si potrà ottenere consenso (parleremo di questo tipo di problema nelle sezioni successive). Un altro metodo, simile alla *proof-of-work*, è la *proof-of-stake*, o *POS*, dove i miner devono provare

all'intero sistema di avere a disposizione un certo capitale, o *stake*. Esistono anche altri tipi di *proof-of-** che vengono adottate da alcune particolari implementazioni di sistemi basati su blockchain. Filecoin [10] ha introdotto la *proof-of-storage*: i miner mettono a disposizione di altri utenti una porzione del loro spazio di memorizzazione, più ne mettono a disposizione e più hanno importanza sulla rete. La *proof-of-storage* viene rinforzata dall'introduzione di altri due meccanismi di conferma, ossia la *proof-of-replication* (verifica se i dati sono stati correttamente replicati nello spazio messo a disposizione dal miner, per evitare che i miner rimuovano le copie di dati dal proprio storage) e la *proof-of-spacetime* (conferma che i miner hanno tenuto memorizzato dei dati per un determinato lasso di tempo).

Le blockchain *permissioned*, invece, non fanno uso del *mining*, perché tutti gli attori che prendono parte al meccanismo di consenso sono noti a priori, per questo usano algoritmi di consenso come quelli spiegati nella sezione precedente.

2.4 Inviolabilità del passato

Come già spiegato in precedenza, le transazioni sulle blockchain vengono inserite in blocchi che, una volta validati dal consenso, vengono aggiunti alla catena di blocchi validati in precedenza. Così facendo si dà la possibilità a tutti i nodi che partecipano alla blockchain di consultare il cosiddetto *ledger*, un libro mastro distribuito formato dai blocchi già validati: tutte le informazioni riguardo una determinata transazione sono pubbliche e facilmente accessibili. Alla luce di quanto appena detto, la domanda sorge spontanea: "E se un utente malintenzionato volesse cambiare in qualche modo ciò che è stato validato dal consenso?". Ad esempio un nodo malevolo potrebbe modificare il destinatario di una transazione così da diventarne il nuovo destinatario e questo, soprattutto in una blockchain che mette a disposizione una criptovaluta come Bitcoin, potrebbe portare a numerosi problemi.

Una prima barriera contro questi attacchi sono i **timestamp** presenti all'in-

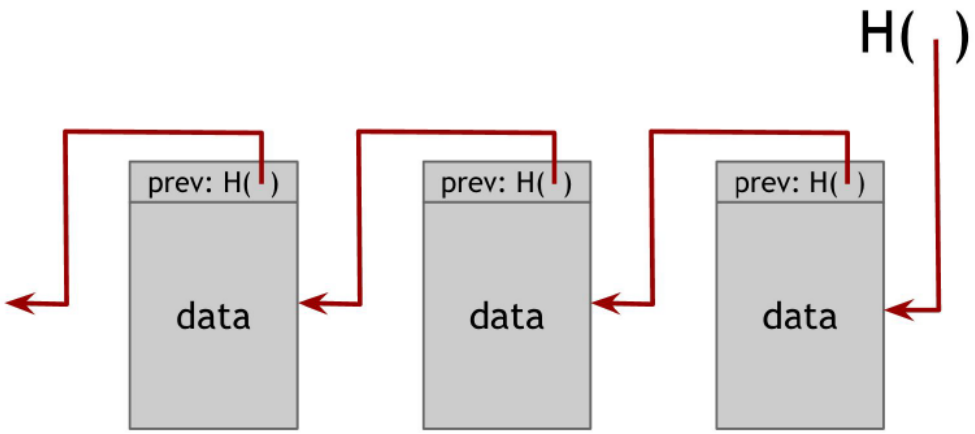

Figura 2.2: Una blockchain è una lista in cui ogni elemento della lista è collegato a quello precedente tramite hash pointer. H() è il puntatore alla testa della lista (blocco più recente) che viene salvato in un altro luogo. [12]

terno di ogni blocco che servono a rendere più difficile questo tipo di attacco. Haber e Stornetta [11] spiegano come creare un timestamp di un documento digitale e come collegare quel documento a un identificativo di chi lo ha rilasciato. All'interno di un sistema basato su blockchain, il timestamp viene inoltre utilizzato dalle **funzioni di hashing**, il cui scopo principale è garantire l'integrità dei blocchi. Ma procediamo con ordine.
Una funzione di *hashing* è una funzione che, a partire da un input, genera una stringa, il cosiddetto *hash* dell'input. Questa funzione rispetta alcune proprietà, quali **determinismo**, **uniformità** e **non invertibilità**:

- **Determinismo**: una funzione di *hashing* genera sempre lo stesso hash in output per un dato input;
- **Uniformità**: ogni valore che forma l'*hash* in output deve essere generato con la stessa probabilità;
- **Non invertibilità**: non è realisticamente possibile ricostruire il dato preso in input a partire dal suo *hash*.

Si analizzi la prima proprietà, ossia il **determinismo**, e cosa vuole dire per la sicurezza di una blockchain. Esistono gli **hash pointer**, ossia un puntatore

che punta al luogo in cui è memorizzata un'informazione e questo puntatore contiene anche un *hash* crittografico dell'informazione a cui punta. La differenza da un normale puntatore è che, grazie all'*hash*, è possibile verificare che l'informazione non sia stata alterata. Infatti, i blocchi della blockchain non hanno un semplice puntatore al blocco precedente (si tenga in mente che nelle liste ogni elemento ha un puntatore all'elemento che lo precede), ma ogni blocco punta al precedente tramite *hash pointer*, mentre l'*hash pointer* della testa della lista, che punta al blocco più recente, verrà memorizzato in un altro luogo.
E sono proprio gli *hash pointer* a garantire l'integrità della blockchain contro possibili attacchi che vanno a modificare i dati presenti nei blocchi. Ad esempio, un attaccante va a modificare dei dati presenti nel blocco k e, siccome i dati nel blocco k sono cambiati, allora l'*hash* salvato nel blocco $k+1$ (che corrisponde all'*hash* del blocco k) non corrisponderà con il nuovo *hash* del blocco k appena modificato; questo perché, grazie alla proprietà del determinismo sopracitata, è statisticamente garantito che l'*hash* del blocco modificato non potrà corrispondere a quello del blocco non modificato e quindi è facile rilevare il cambiamento. Naturalmente l'attaccante potrà risalire la catena e cambiare ogni volta l'*hash* memorizzato in ogni blocco, ma una volta arrivato alla testa della lista si troverà impossibilitato nel continuare con le sue modifiche perché l'*hash pointer* che punta alla testa della lista viene salvato in un posto irraggiungibile dall'attaccante.

Un'altra struttura dati che è possibile costruire tramite l'utilizzo degli *hash pointer* sono i **Merkle Tree** che, come suggerisce il nome, corrispondono a una struttura dati ad albero, più precisamente a un albero binario. Si supponga di avere un numero di blocchi contenenti dei dati, questi blocchi costituiscono le foglie del nostro albero. I blocchi vengono poi raccolti a coppie di due e per ogni paio si va a costruire una struttura dati che ha due *hash pointer*, uno per ogni blocco. Questa struttura dati appena creata corrisponde al livello superiore del nostro albero. La nuova struttura dati verrà poi accoppiata con un'altra che si trova sul suo stesso livello e per ogni paio verrà costruita una nuova struttura dati contenente i loro *hash*. La procedura viene iterata fino ad arrivare ad avere un blocco unico, la *root* dell'albero (Figura 2.4). Il Merkle

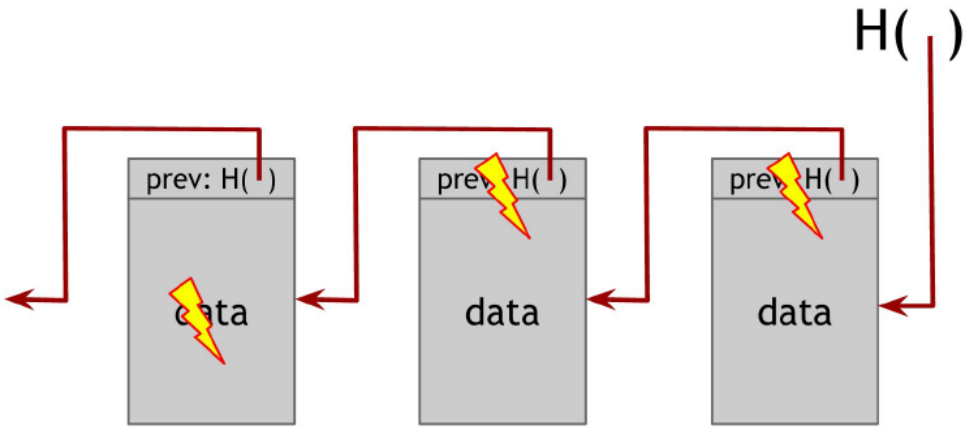

Figura 2.3: Se un attaccante va a modificare un dato in un blocco, allora l'hash pointer del blocco successivo non corrisponderà più. Quindi sarà costretto a risalire la catena, per poi bloccarsi definitivamente alla testa della lista. Questo perché l'hash pointer della testa della lista viene salvato in un luogo irraggiungibile dall'attaccante. [12]

Tree dà la possibilità di riuscire ad arrivare in qualsiasi punto dell'albero semplicemente seguendo gli *hash pointer* fino al blocco di dati desiderato. Anche in questo caso, se un attaccante volesse modificare qualche dato presente in un blocco, si vedrà costretto a modificare tutti gli *hash* dei blocchi di livello superiore, fino ad arrivare all'*hash* della *root* dell'albero che, anche qui, è salvato in un luogo irraggiungibile dall'attaccante.
Un altro pregio dei Merkle Tree è la cosiddetta **proof of membership**: per verificare se un blocco appartiene o meno a un dato Merkle Tree basta conoscere il puntatore alla *root* dell'albero e i blocchi che sono collegati al blocco di dati che vogliamo verificare. È possibile verificare la non appartenenza di un blocco (**proof of non-membership**) usando i **sorted Merkle Tree**, ovvero un Merkle Tree le cui foglie sono disposte secondo un preciso ordine, ad esempio alfabetico o numerico. Il fatto di essere un albero ordinato ci facilita il compito di verificare la non appartenenza: infatti basterà ottenere il percorso dei due blocchi di dati che si dovrebbero trovare prima e dopo il blocco di dati di cui vogliamo verificare la non appartenenza. Se i due blocchi di dati sono consecutivi allora possiamo dire con certezza che il blocco di dati che stiamo verificando non appartiene a quel dato albero [13].

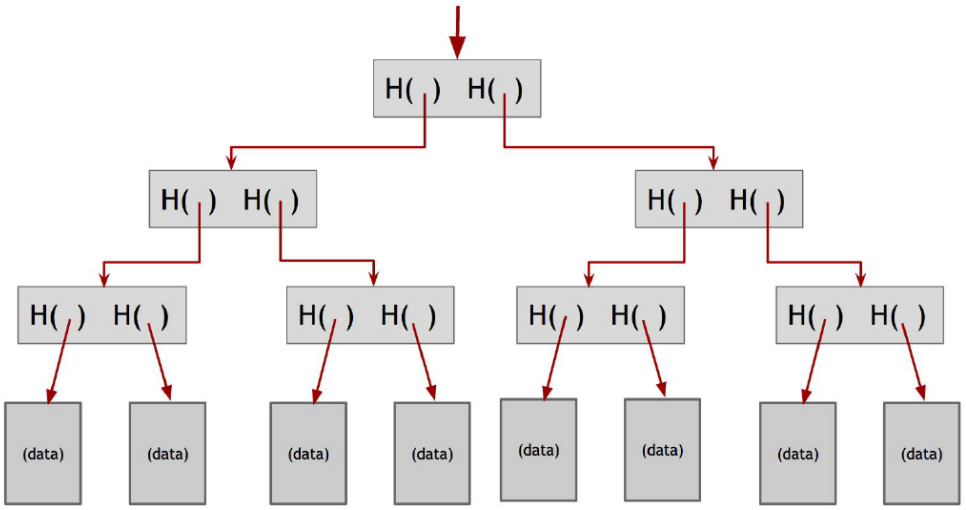

Figura 2.4: In un Merkle tree i blocchi sono raggruppati a coppie e gli hash di ogni blocco viene memorizzato nella struttura dati del livello successivo. [12]

Con queste strutture dati *hash-based* viene garantita la sicurezza e l'integrità dei dati presenti all'interno dei sistemi basati su blockchain. È possibile costruire strutture dati *hash-based* a partire da qualsiasi tipo di struttura dati basata su puntatori, a patto che sia una struttura dati aciclica.

2.5 Attacchi alla blockchain e sicurezza in generale

Precedentemente sono stati già citati alcuni tipi di attacchi e si è analizzato come le blockchain siano riuscite a garantire la sicurezza dei rispettivi nodi. Ma esistono numerosi altri attacchi che sono stati sferrati ai danni delle blockchain nel corso degli anni, alcuni con effetti davvero disastrosi per il sistema.

Un attacco facilmente riproducibile soprattutto nelle blockchain *permissionless* è il famosissimo **attacco Sybil** [14]: in assenza di un ente centrale che gestisce le identità dei vari partecipanti al sistema, un attaccante può creare a

piacimento un numero indefinito di identità, facendole sembrare appartenenti ad altri utenti quando invece vengono controllati dalla stessa entità. Questo tipo di attacco può essere molto pericoloso soprattutto perché potrebbe intaccare l'integrità e l'equità di giudizio del consenso della blockchain. L'attacco può essere prevenuto inserendo alcuni meccanismi che rendano impossibile o più difficile da effettuare; infatti, la già citata *proof of work* e le *Certification Authorities* che gestiscono le identità dei nodi sono le misure di sicurezza adottate dalle moderne blockchain.
Viene da sé che il beneficio di avere un ente centrale che gestisce tutte le identità del sistema rende impossibile l'attacco Sybil, tramite il rilascio di **certificati** firmati digitalmente dall'ente stesso, o come viene comunemente chiamato in ambito informatico **certification authority**, ma questo vale per le blockchain *permissioned*. Per quanto riguarda le blockchain *permsissionless*, la *proof of work*, oltre che per ottenere consenso, è stata introdotta per scoraggiare possibili attacchi Sybil, il perché verrà spiegato nel dettaglio nella prossima sezione.

Si tenga in considerazione, poi, un altro tipo un altro attacco. Ad Alice non va a genio Bob. Alice può decidere di rimuovere ogni transazione di Bob da ogni blocco da lei proposto. Questo tipo di attacco è un classico esempio di attacco di tipo **denial of service**, ma in questo tipo di ambiente non crea altro se non del ritardo nell'accettare la transazione di Bob. Questo perché, anche se la transazione di Bob non viene inclusa nel blocco di Alice, Bob non deve far altro che aspettare che un altro nodo onesto possa proporre il suo blocco con all'interno le transazioni di Bob.

Un altro attacco che caratterizza le blockchain è il **double-spending attack**. Prima di descrivere l'attacco è utile ricordarsi che un asset gestito da una blockchain può essere visto come un *token*. Alice vuole comprare un bene digitale offerto dal sito di Bob. Allora aggiunge al suo carrello l'oggetto che vuole acquistare dal sito di Bob, prepara una transazione per il pagamento e la trasmette alla rete, la transazione verrà poi inserita in un blocco che poi successivamente verrà aggiunto alla blockchain. In una situazione normale il token di Alice viene consumato al momento della transazione, quindi una volta

confermata la transazione il *token* non è più di Alice ma passa nelle mani di Bob. Bob, vedendo che la transazione di Alice è stata validata, manda il bene digitale che Alice aveva messo precedentemente nel carrello. Supponiamo che il nodo successivo che "avrà la parola" sarà proprio Alice. Alice allora potrebbe creare un blocco che andrebbe a invalidare il blocco precedente (quello contenente la transazione Alice →Bob, di fatto annullandola) e poi può includere una transazione che trasferisca il *token* utilizzato per pagare Bob a un altro indirizzo controllato da lei stessa, riappropriandosi del *token*. In questo modo, Bob non riceverà alcun soldo e Alice potrà spendere di nuovo lo stesso *token* che aveva speso per Bob e, siccome due transazioni consumano lo stesso *token*, soltanto una di esse potrà essere inserita nella blockchain. Se Alice riesce a includere il pagamento verso l'altro suo indirizzo allora la transazione Alice →Bob non verrà più considerata perché non potrà più essere inclusa all'interno della blockchain.

Un altro dibattito si è acceso negli ultimi tempi per quanto riguarda le *mining pool*, ossia dei portali dove più *miner* possono collaborare insieme al fine di alleggerire la difficoltà del puzzle da risolvere perché diviso per un numero altissimo di partecipanti e ottenere un profitto, seppur condiviso con l'intera *pool*. È il caso del "**51% attack**": se un ente riesce in qualche modo a ottenere il 51% del potere di una rete può mettere a rischio il consenso e la stabilità della rete stessa. Il fatto che una *mining pool* possa raggiungere il 51% dell'*hashing power* complessivo di una rete può essere un motivo di riflessione in più, visto che con un tale potere si ha la possibilità di "rompere" il consenso, facendo il buono e il cattivo tempo.[15] spiega esattamente cosa può accadere in caso di una situazione del genere.

2.6 Incentivi e disincentivi economici

I *miner* sono il motore delle blockchain *permissionless*: sono loro che mantengono il sistema in condizioni ottimali, sono loro che "minano" i blocchi e creano consenso. Vanno quindi in qualche modo ricompensati per il lavoro che svolgono per il sistema, che altrimenti sarebbe per loro molto sconvenien-

te economicamente parlando. Si analizzi, ad esempio, la blockchain Bitcoin. Un *miner* ottiene una ricompensa per essersi comportato in maniera onesta all'interno del sistema e per essere riuscito a risolvere il puzzle computazionale prima di tutti gli altri. Questo nodo può inserire poi nel blocco appena creato una transazione speciale, una transazione che andrà a creare nuova valuta che verrà assegnata all'indirizzo che il *miner* specificherà nella transazione (naturalmente, la maggior parte dei *miner* scelgono il proprio indirizzo). Questa ricompensa viene chiamata **block reward**. In Bitcoin, il valore di questa ricompensa viene dimezzato ogni 210.000 blocchi che, data la frequenza di creazione dei blocchi in Bitcoin, corrispondono all'incirca a un periodo di tempo di quattro anni. Questo perché, come qualunque altra valuta, le criptovalute non sono infinite, vengono create anche esse e sono un bene finito. Quindi si arriverà in un momento in cui le block rewards saranno talmente tanto basse da quasi considerarsi nulle. E a quel punto cosa spingerà i nodi a continuare a comportarsi in maniera corretta? Le **transaction fees**, che si possono definire definire come vere e proprie "mance". Il creatore di una transazione può scegliere di far sì che il valore dell'output sia inferiore del totale del valore degli input. Quindi, chiunque poi riuscirà a creare il blocco che contiene quella transazione potrà riscuotere la differenza, questo per ogni transazione.

Ma ci sono anche disincentivi che scoraggiano comportamenti scorretti da parte dei nodi. Prima su tutti, la soluzione del puzzle computazionale, ovvero riuscire a ottenere una *proof of work* valida è computazionalmente impegnativa. In soldoni, un nodo deve riuscire a trovare un numero (**nounce**) tale che calcolando l'*hash* della stringa composta da questo numero, l'*hash* del blocco precedente e la lista delle transazioni che compongono il blocco di cui si sta calcolando il *nounce* il numero dell'*hash* di output sia minore di un numero target. Questi *hash puzzles* sono difficili da computare: alla fine del 2014 la difficoltà media di un puzzle era di 10^{20} *hash* per blocco. Quindi un *miner* aveva $1/10^{20}$ di possibilità di trovare il *nounce* giusto. Tenendo in mente l'attacco Sybil descritto nella sezione precedente, il fatto che la *proof of work* sia dispendiosa da computare è un disincentivo per i *miner* malevoli che cercano di effettuare attacchi del genere, perché devono avere a disposizione molto

hardware. La *proof of work* è un puzzle molto difficile da risolvere, computazionalmente parlando, ma triviale da verificare: dopo numerosi tentativi, il nodo riesce a risolvere il puzzle e a trovare il *nounce* che corrisponde alla soluzione del puzzle. Dovrà poi necessariamente pubblicarlo all'interno del blocco. Gli altri nodi, per verificare la correttezza della soluzione, non dovranno far altro che computare l'*hash* dell'intero blocco e verificare che l'output sia inferiore del target.
Un altro disincentivo, questa volta non correlato al sistema stesso, è il costo dell'hardware adatto al mining e i costi di mantenimento, quindi elettricità e raffreddamento del sistema. Chiunque può fare mining se ha a disposizione un computer, ma naturalmente si avranno risultati migliori con hardware migliori, vista la difficoltà computazionale del puzzle. Recentemente, si è passati dal mining tramite CPU al mining tramite GPU, questo perché le GPU hanno più core rispetto alle CPU e riescono quindi a svolgere il compito in maniera più agile, anche se la natura delle GPU non è quella di computare dati matematici ma è quella di elaborare immagini. Per questo, il mercato delle schede video ha subito un'impennata [16] dovuta alla richiesta incessante da parte dei miner che vogliono accaparrarsi gli ultimi modelli. Esistono anche hardware costruiti appositamente per il mining, i cosiddetti **ASIC** (*Application-Specific integrated circuit*) e le blockchain stanno correndo ai ripari iniziando a prediligere puzzle di tipo *memory-hard*, puzzle che hanno bisogno di molta memoria per essere computati invece di aver bisogno di molto tempo di esecuzione di CPU/GPU [12].

Capitolo 3

Smart contracts

In questo capitolo vengono introdotti gli smart contract, elementi caratteristici di alcuni sistemi basati su blockchain. Verranno poi discusse alcune sue potenzialità espresse in varie implementazioni e i problemi e le criticità che sono venuti alla luce durante questo studio.

3.1 Definizione

Gli smart contract sono ancora orfani di definizione formale. L'idea di smart contract viene presentata per la prima volta da Nick Szabo nel 1994 [17], dove Szabo considera i distributori automatici come i diretti discendenti degli *smart contract*: il distributore, una volta che l'utente ha introdotto le monete, riesce a erogare il prodotto e il resto, seguendo uno schema di procedure riconducibile a un **FSA**, *Finite State Automata* o *automa a stati finiti*, ossia un automa in grado di descrivere in maniera formale il comportamento di un sistema [18]. Allo stesso modo, quando uno smart contract viene invocato, quest'ultimo fornisce un risultato deterministico dati in ingresso degli input.

In aggiunta a questo primo concetto di smart contract, negli ultimi anni in molti hanno cercato di dare una definizione quanto più precisa e moderna: se ne possono trovare alcune più recenti che cercano di inquadrare il più possibile il concetto, che di per sé è molto semplice, pur mancando tutt'oggi una definizione formale degli smart contract e della loro semantica operazionale. Uno

spunto interessante, da cui partire ad analizzare alcuni aspetti chiave degli smart contract e iniziare a formalizzare una definizione, viene dato da Gendal [19] ed è il seguente:

A smart-contract is an event-driven program, with state, which runs on a replicated, shared ledger and which can take custody over assets on that ledger.

Scomponendo questa frase, è possibile ricavare alcuni aspetti chiave di uno smart contract. *A smart-contract is an event-driven program*: come si evince da questo estratto, lo smart contract viene identificato come un **programma**, quindi come codice eseguibile e interpretabile da una macchina, non più come un semplice dato passivo facente parte di una determinata blockchain, ma come un programma attivo che risponde a degli stimoli esterni (*event-driven*). Un'altra caratteristica importante, già anticipata nel capitolo precedente di questa tesi, è la presenza di uno **stato** del contratto, che può essere cambiato da transazioni o anche dal contratto stesso. È lecito dire, pertanto, che gli smart contract incapsulano lo stato, similmente agli oggetti nella programmazione orientata a oggetti.
Si può aggiungere altro alla definizione che si sta cercando di creare grazie a Greenspan [20]. Lo smart contract riesce a rappresentare la *business logic* sotto forma di programmi e gli eventi che attivano quella logica vengono rappresentati come messaggi, o più precisamente transazioni, scambiati tra questi programmi e gli utenti finali.
La logica introdotta da un smart contract deve essere rispettata da qualsiasi transizione invochi quel determinato contratto: qualora una transazione non rispettasse ciò che il codice implementa, l'invocazione genererebbe un errore e la transazione non andrebbe a buon fine.

Non esiste un linguaggio di programmazione standard per quanto riguarda gli smart contract: ogni blockchain ha il proprio linguaggio di programmazione messo a disposizione agli sviluppatori, ognuno con pro e contro. Lo smart contract, una volta programmato, deve essere messo sulla blockchain: questo è possibile tramite una transazione di *deploy* e, una volta che il contratto viene correttamente registrato sulla blockchain, esso diventa accessibile a tutti tra-

mite un indirizzo[1] che gli viene assegnato dal protocollo e restituito al mittente come risultato della transazione di deploy.

Riassumendo, lo **smart contract**:

- è un programma che risponde a eventi esterni, con variabili che gestiscono lo stato dello stesso, raggiungibile tramite indirizzo assegnatoli dalla blockchain;
- incarna una logica applicativa e dà una rappresentazione agli eventi che attivano quella logica;
- tramite questa logica, valida le transazioni che invocano quel contratto;
- è programmabile utilizzando diversi linguaggi di programmazione, in base all'implementazione della blockchain.

C'è inoltre in atto un dibattito sulla questione della validità legale di uno smart contract in sede di tribunale. Anche se uno smart contract può modellare delle leggi o garantire le clausole contrattuali, in tribunale un codice di un programma può risultare non comprensibile da tutti. Questa situazione porta a porsi una domanda: "Si può sviluppare uno smart contract in modo tale da essere compreso anche da chiunque?". Una risposta è il cosiddetto **Ricardian contract**. Proposti da Iam Grigg [21], i *Ricardian contracts* incarnano l'idea di scrivere un documento che sia comprensibile e accettabile sia da una macchina che da un tribunale. Un *Ricarian contract* gode delle seguenti proprietà:

- un contratto rilasciato da un ente;
- è facilmente leggibile da chiunque, come se fosse un contratto cartaceo;
- è interpretabile da una macchina;
- è firmato digitalmente;
- ogni contratto possiede un identificatore univoco.

[1] ricordiamo che gli smart contract sono impiegati in blockchain con modello *account-based*, dove ogni ente è mappato da un indirizzo

Questo tipo di contratto corrisponde ha un documento che contiene i termini contrattuali, scritti in linguaggio legale insieme a delle parole chiave comprensibili dalle macchine. Il documento viene poi firmato digitalmente da chi ha creato il contratto usando la sua chiave privata. Una volta firmato, viene generato il suo hash che lo identificherà e verrà usato e firmato dalle parti che useranno quel contratto come riferimento a una transazione.
Un *Ricardian contract* si differenzia da un normale smart contract per il fatto che quest'ultimo non contiene alcun documento contrattuale e il suo scopo è principalmente quello di eseguire codice che rispecchi il contratto. Al contrario, il *Riccardian contract* pone la sua attenzione alla ricchezza semantica. Questa semantica può essere divisa in due parti: semantica operazionale e semantica delle denotazioni. La semantica operazionale va a definire l'esecuzione del contratto, mentre la seconda va a "interfacciare" il contratto con il mondo reale.

3.2 Relazione con State Machine Replication

Come già detto in precedenza, gli smart contract possono essere considerati come dei veri e propri programmi che vengono inseriti all'interno della blockchain, in modo da permettere l'interazione con l'intero sistema. Data la loro natura, permettono di effettuare **computazioni deterministiche** da parte di tutta la rete: a parità di input si avranno sempre gli stessi output da parte di ogni nodo. Il principio di replicare degli elementi che computano in maniera deterministica su più nodi di una rete viene chiamato in informatica **State Machine Replication** o **State Machine Approach** [22].

Con un approccio del genere, si dà la possibilità a un sistema di computare copie di stati identiche e garantire un certo grado di *fault-tollerance*: infatti, il sistema continuerà a operare anche nel caso in cui alcuni server (o nodi) vadano offline o si comportino in maniera non consona al comportamento dell'intera rete. Ogni server contiene al suo interno un registro con una serie di istruzioni che le macchine a stati devono seguire in ordine. Anche qui, le macchine a stati permettono una computazione deterministica, quindi, a un dato

input corrisponderà inequivocabilmente lo stesso output per tutte le macchine a stati presenti nel sistema.

Il modello permette, come già detto in precedenza, al sistema di avere un certo grado di fault tollerance. Trattandosi di macchine con una computazione deterministica, risulterà particolarmente facile identificare quali nodi della rete devono essere considerati come *faulty*, dove faulty indica un errore di computazione, quindi un problema per quanto riguarda la computazione in sé, oppure un comportamento diverso da quello che ci si aspetterebbe. Schneider [23] divide la questione *faulty* in due classi:

- **Byzantine failures**, traducibile in "fallimenti bizantini", ossia il nodo si comporta in maniera errata a causa di una possibile manomissione da parte di utenti malevoli;
- **Fail-stop failures**, traducibile in "fallimenti bloccanti", dove il nodo, a seguito di un fallimento, entra in uno stato che permette agli altri nodi di percepire che è avvenuto un fallimento e successivamente si blocca.

Per mantenere la fault tollerance, inoltre, tutti i nodi devono ricevere tutte le transazioni di stato e tutte nello stesso ordine. Queste due proprietà sono soddisfacibili solo in caso di presenza di un algoritmo che permetta a tutte le repliche di ricevere aggiornamenti consistenti e ordinati sugli stati. Per questo, è necessario inserire all'interno del sistema un algoritmo di **consenso**, permettendo al sistema di gestire in maniera corretta i vari fallimenti dei nodi. Tutti questi elementi appena analizzati vengono messi a disposizione dalla tecnologia blockchain, facilitandone l'utilizzo a chi vuole basare i propri sistemi su questa nuova tecnologia.

3.3 Caratteristiche generali

Nella Sezione 3.1 si parla di transazioni che permettono il passaggio da uno stato all'altro di uno smart contract. Ma in cosa consistono queste transazioni e come vengono attivate dagli altri nodi? Allo stato attuale dei vari sistemi, le

transazioni non sono altro che messaggi firmati dal creatore del messaggio diretti allo smart contract. Questi messaggi sono delle chiamate a delle funzioni del contratto stesso, che provocano un cambiamento dello stato del sistema.
Viene naturale paragonare, quindi, uno smart contract a un oggetto remoto, o **remote object**. Un oggetto è, in un linguaggio di programmazione a oggetti, un'istanza di una classe, che contiene funzioni e variabili proprie, ed è posto all'interno di un server remoto. L'oggetto è poi accessibile tramite indirizzo e le sue funzioni sono utilizzabili tramite **Remote Procedure Call**, ovvero vengono invocate dal altri nodi tramite messaggi che viaggiano sulla rete.

Se consideriamo quindi gli smart contract come oggetti remoti, dobbiamo analizzare alcuni concetti. Primo su tutti, come un oggetto remoto viene inizializzato, ossia come uno smart contract viene inserito all'interno della blockchain. Ogni tecnologia di blockchain ha il proprio metodo per effettuare il deploy di un contratto. Ad esempio, in Ethereum un contratto può essere essere inserito all'interno della blockchain tramite strumenti come Mist[2], che facilita e automatizza le procedure di deploy di uno smart contract: una volta scritto il proprio contratto, solitamente scritto usando il linguaggio Solidity[3], Mist permette d'interagire con la blockchain Ethereum e dà la possibilità a chiunque di effettuare il deploy di uno smart contract all'interno di essa. Ma a prescindere dalla blockchain di riferimento, uno smart contract sarà sempre identificato da una stringa, o indirizzo, che lo rende raggiungibile da qualunque nodo all'interno della rete. L'indirizzo viene assegnato allo smart contract non appena il deploy va a buon fine. Una volta pubblicato il contratto sulla blockchain, l'indirizzo corrispondente viene inserito all'interno dello stato globale del sistema, dove sono presenti tutti gli indirizzi delle entità che fanno parte del sistema. A partire da questo indirizzo, è possibile interagire con lo smart contract tramite chiamate remote.
Ci sono, inoltre, altre caratteristiche che accomunano gli smart contract agli oggetti. Una di questa è che entrambi incapsulano lo stato e il comportamento, ma sia gli smart contract sia gli oggetti non incapsulano il flusso di controllo.

[2] `https://github.com/ethereum/mist`
[3] `https://github.com/ethereum/solidity`

3.4 Criticità

Ci sono alcuni aspetti delle interazioni con gli smart contract che devono essere puntualizzati. Ad esempio, è possibile per uno smart contract comunicare con un altro smart contract? Ed è possibile per un'entità esterna alla blockchain comunicare con uno smart contract presente nella blockchain e viceversa? La risposta a entrambe le domande è "sì".

Gli smart contract hanno la possibilità di comunicare con altri contratti della blockchain nella stessa maniera con cui i nodi comunicano con gli smart contract, ossia tramite indirizzo. Ma in questo contesto il codice che effettua la chiamata a un altro contratto deve essere inserito all'interno del contratto chiamante, perché gli smart contract, una volta effettuato il loro deploy sulla blockchain, non possono essere modificati. Si analizzerà ulteriormente cosa comporta il fatto che gli smart contract siano oggetti immutabili nella Sezione 3.5. Ci sono due principali metodi per effettuare una comunicazione tra contratti: tramite chiamate a funzioni primitive del linguaggio di programmazione del contratto, o tramite quelli che vengono chiamati *upgradeable contracts*, ma questi ultimi verranno analizzati in dettaglio nella Sezione 3.5.

Come già detto in precedenza, l'uso di chiamate a funzioni primitive è strettamente dipendente dal linguaggio di programmazione per smart contract che viene utilizzato in una determinata blockchain. Solidity, ad esempio, permette di instaurare una comunicazione tra due smart contract in maniera immediata grazie all'utilizzo di due funzioni primitive, *call* e *delegateCall*. La prima permette a un contratto di chiamare una funzione di un altro contratto ed eseguire il suo codice. La primitiva *delegateCall* è simile alla *call*, con l'unica differenza che l'esecuzione del codice non viene effettuata sullo stato del contratto chiamato, ma sullo stato del contratto chiamante: praticamente, il contratto chiamante mette a disposizione il codice che andrà poi a variare lo stato del contratto chiamante. Così facendo, si dà la possibilità a un contratto esterno di eseguire codice di cui lo sviluppatore del contratto chiamante potrebbe non prevedere il comportamento, sia in termini di computazione, sia in termini di

sicurezza. Sia *call* che *delegateCall* non hanno un valore di ritorno a causa delle limitazioni della EVM, la virtual machine usata da Ethereum.

L'interazione con gli smart contract da parte di entità esterne alla blockchain viene resa possibile grazie all'utilizzo degli **oracle**. Gli oracle sono degli agenti che identificano e verificano dati provenienti da sorgenti esterne e li invia alla blockchain per essere usate dagli smart contract. Questi oracle possono essere di quattro tipi: troviamo i **software oracles**, che estraggono le informazioni disponibili sulla rete Internet e le inviano agli smart contract, gli **hardware oracles**, che possono essere sensori che inviano dati in tempo reale a un contratto, gli **inbound oracles**, che inviano dati esterni alla blockchain allo smart contract, e gli **outbound oracles**, che danno la possibilità agli smart contract di inviare dati dalla blockchain al mondo esterno.

3.5 Problematiche

Anche gli smart contract hanno delle problematiche che li caratterizzano. Come già anticipato nella Sezione 3.4, gli smart contract, una volta inseriti all'interno della blockchain, sono immutabili. Quindi, in caso venga rilasciato un contratto contenente un qualche errore di programmazione da parte dello sviluppatore, non sarà possibile correggerlo. Una possibile soluzione a questo problema è utilizzare i cosiddetti **upgradeable contracts**. Questo tipo di contratto separa la business logic dai dati associati alla logica in questione.
L'approccio è molto semplice. Si va a creare uno smart contract che funga da "registro per i contratti": all'interno di questo contratto si va poi a creare una struttura dati che contiene le varie coppie nome contratto - indirizzo. Se si vuole modificare un contratto, basterà rieffettuare il deploy dello smart contract contenente la business logic e poi sostituire l'indirizzo corrispondente a contratto con quello del nuovo contratto all'interno del "registro". Così facendo è possibile "modificare" la business logic di un contratto già presente nella blockchain.

Gli smart contract sono anch'essi vulnerabili ad attacchi da parte di utenti

malintenzionati. Nella storia della sicurezza degli smart contract c'è stato un avvenimento che ha provocato un forte disordine all'interno del mondo delle blockchain. Stiamo parlando del famigerato "**DAO Attack**", che ha fruttato all'attaccante ben 3,6 milioni di ether (l'equivalente di 50 milioni di dollari al cambio di allora) [24], ed è stato sferrato ai danni della blockchain Ethereum. Il **DAO**, o **Decentralized Autonomous Organization**, è stato uno dei progetti di crowdfunding più finanziati di sempre [25] ed è stato avviato nell'aprile del 2016. L'attacco è stato effettuato nel giugno dello stesso anno. Prima di scendere nei dettagli dell'attacco, è necessario introdurre alcuni concetti di sicurezza propri di Ethereum.

Come già spiegato in precedenza, gli smart contract di Ethereum sono scritti con un linguaggio di programmazione proprio, Solidity. Il codice scritto in Solidity deve però essere compilato in bytecode prima di essere inviato alla blockchain. Successivamente, questo bytecode potrà essere eseguito correttamente dalla virtual machine di Ethereum, la EVM. Una funzione di un contratto possono essere eseguite tramite l'invio di un messaggio, contenente tutti i dati necessari alla chiamata. Solidity mette a disposizione un tipo di funzione particolare, la cosiddetta **fallback function**, che viene eseguita in tre casi: quando la funzione chiamata dal messaggio non corrisponde a nessuna delle funzioni presenti nel contratto, quando un contratto riceve della valuta (l'*ether*) senza ulteriori dati e quando non sono presenti dati all'interno della chiamata. L'attaccante aveva creato un contratto malevolo che andava a sfruttare il comportamento delle fallback functions e di aclune primitive di Solidity, permettendogli di ottenere una tale somma in pochissimo tempo. Una panoramica molto interessante sugli attacchi effettuabili ai danni della blockchain Ethereum viene effettuata da Atzei, Bartoletti e Cimoli[26].

Capitolo 4

Confronto tra tecnologie esistenti

In questo capitolo si effettua un confronto tra tecnologie blockchain che hanno caratteristiche peculiari e interessanti, quali *Hyperledger Fabric* di IBM, *Corda* di R3 e *Tendermint* di Cosmos. Vengono scelti dei punti di confronto tra le tre tecnologie che sono spiegati in maniera esaustiva.

4.1 Tecnologie blockchain a confronto

Esistono numerose implementazioni della tecnologia blockchain, la maggior parte delle quali riguarda la creazione di blockchain a supporto di criptovalute, come le già citate Bitcoin ed Ethereum. Ma molte sono le idee innovative che ruotano intorno a questo ecosistema che non incentrano il loro focus sulle criptovalute. Ad esempio, Filecoin [10] permette la creazione di un sistema di memorizzazione cloud decentralizzato che si basa su blockchain. Un'altra tecnologia innovativa viene presentata dalla già citata Blockstack [2], che si pone l'obiettivo di voler reingegnerizzare il web in maniera del tutto decentralizzata, tramite l'utilizzo della tecnologia blockchain e delle reti peer-to-peer.

Le tecnologie che verranno analizzate sono delle blockchain che mettono a disposizione strumenti e caratteristiche molto interessanti per lo sviluppo futuro di questo settore. Sono **Hyperledger Fabric**, tecnologia blockchain sviluppa-

ta da IBM per le aziende, **Corda**, creata da R3 e infine abbiamo **Tendermint**, curata da Cosmos.

I criteri con cui andremo a confrontare queste tecnologie blockchain sono principalmente tre:

- come viene gestito lo stato all'interno della blockchain;
- come viene gestito il consenso all'interno della blockchain;
- come vengono gestiti gli utenti e la loro identità all'interno della blockchain.

Verrà inoltre analizzato come vengono declinati gli smart contract all'interno delle sopracitate tecnologie.

4.1.1 Hyperledger Fabric

Hyperledger Fabric [27], che fa parte del progetto collaborativo Hyperledger nato a dicembre del 2015 e voluto da Linux Foundation, è una piattaforma per lo creazione di sistemi distribuiti promosso da IBM per soluzioni private.

Hyperledger Fabric è una blockchain privata, quindi *permissioned*, dove è presente una certification authority, cioè un'ente che gestisce i permessi dei vari membri della rete, chiamata **Membership Service Provider** (**MSP**). Ogni MSP può definire il proprio concetto di identità e le regole che queste devono rispettare per essere ritenute valide (*identity validation*). Inoltre, queste regole vengono usate per l'autenticazione dei membri durante le varie comunicazioni (*signature generation and verification*). Una blockchain basata su Hyperledger Fabric può avere più di un nodo MPS, garantendo modularità e interoperabilità tra diversi standard di autenticazione.

Hyperledger Fabric è molto flessibile: i dati possono essere memorizzati in diversi formati ed è compatibile con numerosi algoritmi di consenso. Dà anche la possibilità di creare **canali**, permettendo a un gruppo di membri del

sistema di creare uno spazio privato dove memorizzare le proprie transazioni che non si vogliono trasmettere agli altri partecipanti al sistema. Questo trova facile impiego in una blockchain composta da aziende dove due o più di esse possono trovare un accordo in privato senza che gli altri membri lo sappiano, per evitare la fuga di informazioni sensibili.

Il sistema core di Hyperledger Fabric è composto da due parti: lo stato globale (**world state**) e il registro delle transazioni (**transaction log**). Ogni partecipante al sistema ha una copia del contenuto della blockchain di cui fa parte. Lo stato globale descrive lo stato del *ledger* in un determinato istante di tempo, mentre il registro delle transazioni contiene tutte le transazioni che hanno contribuito a generare l'attuale stato del sistema.

Passiamo agli smart contract. In Hyperledger Fabric vengono chiamati **chaincode**, scritti in Go[1], che implementano un'interfaccia per le applicazioni esterne alla blockchain che vogliono interfacciarsi con essa. Il chaincode viene eseguito in un ambiente circoscritto, chiamato Docker [28], e inizializza e gestisce lo stato del *ledger* attraverso le transazioni inviate dalle applicazioni: ogni chaincode può accedere soltanto agli stati che ha creato, ma è possibile permettere ad altri chaincode, presenti sulla stessa rete, di modificarli tramite appositi permessi.

Per quanto riguarda il consenso in Hyperledger Fabric, non esiste un unico algoritmo uguale per tutte le istanze di questo sistema, ma ognuna può specificare quale utilizzare. Nonostante ciò, esiste una procedura di validazione in comune a tutte le implementazioni di Hyperledger Fabric. Facciamo un esempio. Ci sono due utenti, A e B, che vogliono scambiarsi la proprietà di un oggetto. Entrambi dispongono di un *peer* all'interno della rete dal quale invieranno le transazioni e interagiranno con la blockchain. Su questi peer troviamo installato il chaincode al cui interno è presente la logica dello scambio, insieme alle regole di convalida, dette *endorsement policy*, per quel chaincode. L'uten-

[1]IBM ha intenzione di ampliare il supporto ad altri linguaggi di programmazione come il Java

te *A* invia la transazione che avvia lo scambio. Questa viene successivamente inoltrata ai nodi che devono convalidarla. Per comodità d'esempio, i nodi che dovranno validare la transazione sono i peer corrispondenti ai due clienti, quindi la transazione viene inviata ai due peer. Una volta approvata la transazione (*transaction endorsing*), viene creata una proposta di transazione (*transaction proposal*). Quest'ultima corrisponde a una richiesta di invocazione di una determinata funzione del chaincode che permetterà, una volta completato il processo di consenso, di scrivere e/o leggere dati sul *ledger*. Questa funzione usa anche le credenziali crittografiche dell'utente per creare una firma digitale per la proposta, che verrà poi inoltrata ai nodi validatori (*endorsing peers* o *endorsers*) di quel chaincode. I peer controlleranno che la proposta di transazione sia conforme alle regole, che non sia già stata trasmessa in precedenza, che le firme siano valide, contattando il MSP, e che chi ha inviato la proposta sia autorizzato a effettuare l'operazione richiesta. A questo punto, se la proposta viene ritenuta valida, i peer eseguono il chaincode della richiesta, passando come input i dati presenti all'interno della proposta. Il risultato di questa chiamata conterrà un valore di risposta (*response value*), un *read set* e un *write set*, ma ancora non è stata fatta alcuna modifica al *ledger*. Questi valori vengono rimandati indietro all'applicazione, che controllerà se le risposte dei peer validatori sono identiche. Se l'applicazione, successivamente, dovrà aggiornare lo stato del *ledger* allora invierà la transazione al sistema di ordinamento delle transazioni (*Ordering Service*), il quale riceve le transazioni di tutti i canali presenti sulla rete e le ordina cronologicamente e crea, per ogni canale, i blocchi contenenti le transazioni. Questi blocchi appena creati vengono consegnati a tutti i peer presenti sulla rete e aggiungeranno il blocco alla propria catena. La Figura 4.1 raffigura in maniera semplificata la comunicazione tra due enti.

4.1.2 Corda

Corda [30], una tecnologia sviluppata da R3[2], trova facile impiego all'interno dell'ambiente finanziario, viste le caratteristiche delle sue componenti chiave.

[2] `https://www.r3.com/`

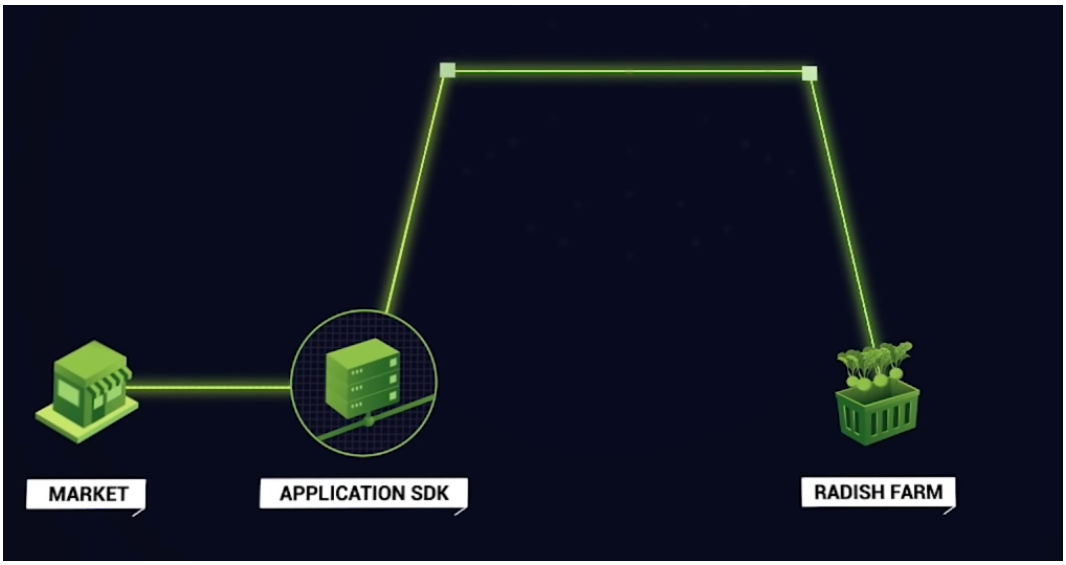

Figura 4.1: Schema raffigurante l'ecosistema di Hyperledger Fabric. Un ente comunica con un altro ente tramite i loro peer presenti sulla rete Hyperledger Fabric, rappresentati in figura da quadratini verdi. L'ente che avvia la comunicazione invia una transazione che dovrà essere validata. Una volta validata, la transazione verrà inviata nuovamente all'applicazione, che a sua volta la invierà al servizio di ordinamento, o consenso. Una volta ottenuto il consenso su quelle transazioni, il blocco che conterrà quelle transazioni verrà inviato ai peer e aggiunto al loro ledger. [29]

Corda è una tecnologia blockchain *permissioned* composta da una rete di nodi P2P. Al suo interno troviamo un *network map service* che pubblica l'IP al quale ogni nodo può essere raggiunto (una sorta di servizio di pagine bianche), insieme al certificato d'identità di quel nodo e a un elenco di servizi che offre. Ogni rete Corda ha un ente che regola l'accesso alla rete da parte di nuovi nodi ed è il "**doorman**". I nodi che vogliono entrare a far parte della rete devono fornire delle informazioni al doorman. Se queste informazioni sono sufficienti e vengono ritenute valide dal doorman, il nodo riceve un certificato TLS firmato dalla root-authority che dovrà essere utilizzato per effettuare comunicazioni sulla rete.

In Corda ogni nodo conosce soltanto la porzione dei fatti che lo riguarda direttamente, nessun nodo conosce il *ledger* nella sua interezza. I fatti conosciuti possono essere condivisi con più nodi. L'oggetto fondamentale del concetto di Corda è lo **state object**. Uno stato corrisponde a un oggetto immutabile che rappresenta un fatto conosciuto da uno o più nodi Corda in un dato momento di tempo. Gli stati possono contenere dei dati arbitrari, quindi possono modellare qualsiasi tipo di fatto. Essendo immutabili, non possono essere direttamente modificati per rispecchiare un cambiamento nello stato globale. Corda implementa una linea temporale che contiene tutti i cambiamenti che ha subito uno stato, andando a creare una sequenza dello stato (**state sequence**): quando uno stato deve essere aggiornato, viene creata una nuova versione dello stato e viene aggiunta alla sequenza delle versioni precedenti di quello stato. All'interno di ogni nodo troviamo una raccolta di tutte le sequenze degli stati di cui il nodo è al corrente e viene chiamata **vault**. Naturalmente, l'unica versione dei fatti valida è quella più recente.

Gli smart contract in Corda possono essere considerati come contratti veri e propri. Corda aggiunge un nuovo tipo di validità per quanto riguarda le transazioni, ossia che ogni transazione per essere considerata valida, oltre ad avere tutte le firme necessarie, deve essere contrattualmente valida: ogni stato deve essere collegato a un contratto, il contratto prende in input delle tran-

sazioni e controlla se sono considerate valide in base alle regole del contratto stesso. Una transazione viene considerata valida se viene considerata tale dal contratto di ogni stato preso in input e in output. La figura 4.2 rappresenta in maniera schematica il processo di validazione di una transazione da parte di uno smart contract.

A differenza delle altre tecnologie blockchain, Corda non invia messaggi a tutta la rete, ma usa una messaggistica *point-to-point*. Questo implica che, a ogni proposta di aggiornamento del *ledger*, tutti i partecipanti debbano sapere esattamente quali informazioni mandare, a chi mandarle e in quale ordine. Corda rende automatico questo procedimento grazie all'impiego dei **flow**, ossia una sequenza di passaggi che dice al nodo in che modo effettuare una determinata operazione.

Per quanto riguarda il consenso, in Corda viene diviso in due parti: il **validity consensus** e il **uniqueness consensus**.
Il *validity consensus* è il processo che controlla se vengono rispettate queste due condizioni: la transazione deve essere ritenuta valida dai contratti di ogni stato in input e in output e deve avere le firme di ogni partecipante. Però non basta verificare la validità solo della transazione proposta, deve essere verificata la validità di ogni transazione che ha creato quegli input.
Lo *uniqueness consensus* viene applicato per verificare che un input di una transazione proposta non sia stato consumato da un'altra transazione, evitando quindi il double spend attack. Questo tipo di consenso viene garantito da nodi che possono essere considerati dei veri e propri notai (**notaries nodes**). Questi nodi hanno l'ultima parola per quanto riguarda l'aggiunta di una transazione al *ledger*, firmandole o rifiutandole. Anche qui, come in Hyperledger, non vi è un algoritmo di consenso standard, ma ogni nodo notarile può scegliere il proprio algoritmo in base ai propri requisiti. Corda permette ai suddetti nodi di prendersi carico anche del validity consensus.

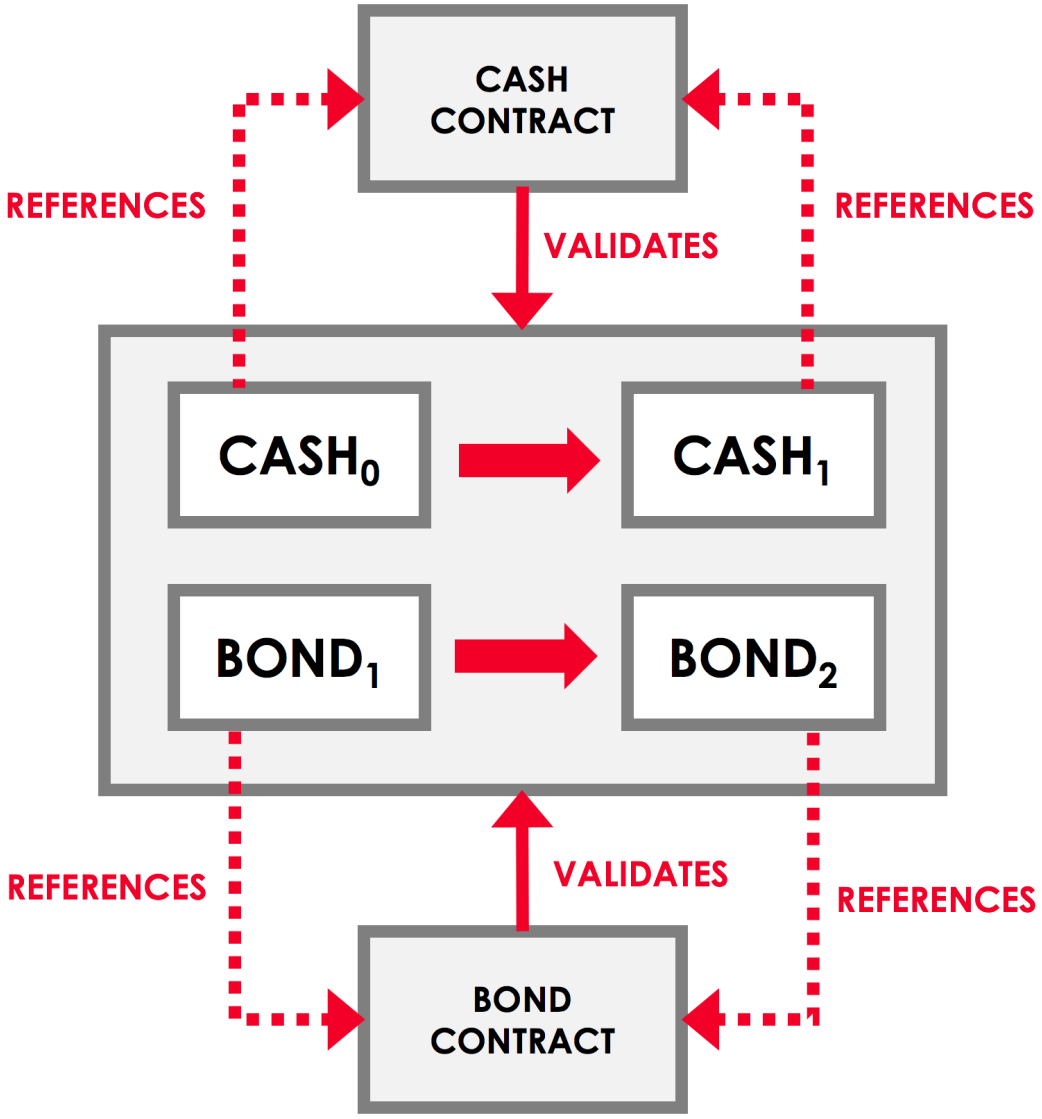

Figura 4.2: Schema raffigurante la validazione di una transazione in Corda. Il passaggio dallo stato iniziale a quello successivo ($Cash_0 \rightarrow Cash_1$) viene regolato dal contratto a cui gli stati si riferiscono. Nel caso degli stati Cash, il contratto di riferimento è identificato da Cash Contract e tutte le transazioni di uno stato Cash devono necessariamente far riferimento a quel contratto. [30]

4.1.3 Tendermint

Passiamo a Tendermint [31]. Sviluppata da Cosmos[3], permette la replicazione di un'applicazione su più macchine seguendo il paradigma della tecnologia blockchain con un algoritmo di consenso *Byzantine Fault Tollerant*.

Tendermint è divisibile in due parti principali: **Tendermint Core**, ossia la parte che gestisce il "motore" della blockchain, e l'**Application BlockChain Interface** (**ABCI**), che permette alle transazioni di essere gestite da una logica applicativa scritta in qualunque linguaggio di programmazione. La parte core di Tendermint garantisce, invece, che le transazioni vengano memorizzate su ogni macchina nello stesso ordine.

Analizziamo per prima la parte che si occupa del consenso. Tendermint mette a disposizione dei nodi validatori (**validators**), identificati dalla loro chiave pubblica, e ogni nodo è responsabile del mantenimento di una copia integrale dello stato del sistema, della proposta di nuovi blocchi e del voto per validare i suddetti blocchi. A ogni blocco viene assegnato un indice incrementale (**height**), così facendo si avrà un blocco valido per ogni *height*. Ogni blocco viene proposto da un nodo diverso ogni volta (il nodo viene detto *proposer*), dividendo il processo di consenso in veri e propri round.
Il processo di consenso può essere diviso in 3 fasi:

- **Proposta** (*proposal*): il *proposer* di turno propone un nuovo blocco e gli altri validatori lo ricevono. Se non lo ricevono entro un determinato periodo di tempo si passa al *proposer* successivo;
- **Votazione** (*votes*): la fase di votazione si suddivide anch'essa in due sotto parti, ossia *pre-vote* e *pre-commit*.
- **Lock**: Tendermint si assicura che nessun validatore inserisca più di un blocco a un dato indice (*height*).

Ogni round inizia con una nuova proposta. Il nodo che effettua la proposta prende le transazioni presenti all'interno della sua cache, chiamata **Mempool**,

[3] `https://cosmos.network/`

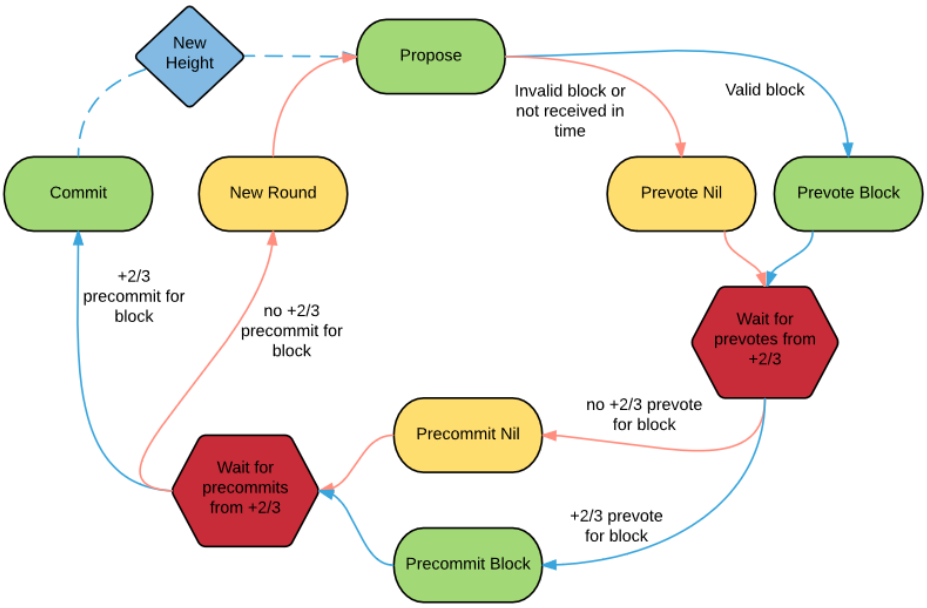

Figura 4.3: Schema raffigurante il sistema di consenso di Tendermint. Si parte con una proposta di un nuovo blocco da parte di un Proposer. Se la proposta viene considerata non valida o non viene ricevuta in tempo, il blocco viene scartato. Si passa poi alla procedura di pre-commit. Se i 2/3 della rete vota a favore del nuovo blocco, quest'ultimo verrà aggiunto alla blockchain. [31]

assembla il blocco e lo spedisce sulla rete tramite un messaggio firmato (*ProposalMsg*). Una volta che la proposta viene ricevuta da un nodo validatore, quest'ultimo firma un messaggio per il *pre-vote* di quella proposta e lo invia a tutta la rete. Se un validatore non riceve una proposta entro un determinato periodo di tempo (*ProposalTimeout*), il suo voto verrà considerato come *nil*. Se almeno i 2/3 della rete hanno votato a favore del blocco, si passa a un'altra votazione, ossia quella per la *pre-commit*. In sintesi, la votazione di *pre-vote* prepara la rete a ricevere un nuovo blocco da inserire alla blockchain. Se la rete è pronta a ricevere questo nuovo blocco, ossia è stato votato dai 2/3 della rete, allora si passa alla votazione per il *pre-commit* e se un validatore riceve voti da almeno i 2/3 dei nodi, il blocco viene aggiunto alla blockchain e viene computato il nuovo stato del sistema. La Figura 4.3 rappresenta in maniera schematica il processo appena descritto.

L'altra componente di Tendermint, l'**Application BlockChain Interface**, funge da interfaccia tra il processo applicativo e il processo di consenso. L'ABCI comunica con il core di Tendermint principalmente attraverso 3 tipi di messaggi: **DeliverTx**, **CheckTx** e **Commit**. Il messaggio **DeliverTx** viene usato ogni qual volta viene inviata una transazione sulla blockchain. L'applicazione poi dovrà verificare la validità della transazione rispetto allo stato attuale. Se la transazione risulterà valida, allora l'applicazione dovrà aggiornare il proprio stato con le nuove informazioni ottenute dalla transazione.
Quando un'applicazione presente sulla rete Tendermint vuole inviare una transazione, i dati immessi dall'utente vengono pre-processati e temporaneamente salvati all'interno di una memoria cache, la **Mempool Cache**. Prima di essere immessa nella **Mempool** vera e propria, ossia la memoria da cui poi un nodo può recuperare le transazioni da includere nel blocco che proporrà, il nodo verifica la validità della transazione tramite **CheckTx**: il contenuto della transazione viene confrontato con l'attuale stato del sistema e, se viene ritenuto coerente con esso, la transazione verrà accettata dal sistema, altrimenti verrà rifiutata. Di primo acchito può sembrare che **CheckTx** sia una **DeliverTx** semplificata, ma in realtà questi due messaggi vengono inviati in momenti diversi. Occorre analizzare le connessioni che l'interfaccia ABCI mantiene per

capire la differenza tra i due. L'applicazione ABCI presente su ogni nodo mantiene tre connessioni con il Tendermint Core: la **Mempool Connection**, usata per validare le transazioni presenti sulla Mempool tramite l'utilizzo di CheckTX, la **Consensus Connection**, usata soltanto quando viene effettuata la commit di un nuovo blocco, e la **Query Connection**, usata per effettuare query all'applicazione senza passare dal consenso. Lo stato dell'applicazione fornisce le informazioni necessarie, solo in lettura, alla Mempool Connection e alla Query Connection, mentre la scrittura viene presa in carico dalla Consensus Connection ed è proprio il blocco ricevuto da questa connessione a contenere tanti messaggi DeliverTX quante transazioni sono presenti nel blocco.
Riassumendo, il messaggio CheckTx viene utilizzato quando una transazione deve essere inserita all'interno della Mempool, quindi prima del processo di consenso, mentre il messaggio DeliverTx viene usato dal consenso quando va ad assemblare il blocco, ordinando le transazioni.
Infine, il messaggio **Commit** viene utilizzato per computare l'hash del Merkle tree corrispondente allo stato dell'applicazione.

4.2 Riepilogo

Nelle sezioni precedenti si è descritto in maniera estensiva ogni aspetto chiave delle varie implementazioni sopracitate. Di seguito, le Figure 4.4 e 4.5 riassumono i risultati di questa analisi.

	Gestione utenti	**Gestione consenso**
Hyperledger Fabric	Permissioned, identità e permessi gestiti dal MSP.	Non esiste algoritmo di consenso specifico, ogni istanza può scegliere quale algorimo usare. Esiste un servizio di ordinamento delle transazioni che genera consenso nella blockchain.
Corda	Permissioned, esiste "doorman" che regola ingresso nodi, root-authority firma un certificato TLS che consegna ai nodi della rete, usato per comunicare.	Validity consensus (verifica la validità della transazione rispetto al contratto di riferimento e verifica che siano presenti le firme dei partecipanti alla transazione) e lo uniqueness consensus (verifica che gli input di una transazione non siano stati già utilizzati in altre transazioni.
Tendermint	Non c'è una vera e propria gestione degli utenti e dei permessi perché Tendermint si occupa principalmente della parte di consenso.	Consenso di tipo BFT. Diviso in tre parti: proposta di un nuovo blocco, votazione del nuovo blocco e lock dell'indice del blocco. Durante la votazione, se i $2/3$ della rete votano a favore, si può procedere. Altrimenti il blocco viene scartato.

Figura 4.4: Tabella riassuntiva del confronto della gestione degli utenti e del consenso tra Hyperledger Fabric, Corda e Tendermint.

	Gestione stato	**Smart contracts**
Hyperledger Fabric	Dati rappresentati nel formato coppia chiave-valore.	In Hyperledger Fabric gli smart contract vengono denominati Chaincode.
Corda	Oggetto immutabile che contiene dati arbitrari, presenza della sequenza degli stati passati.	Possono essere considerati come dei veri e propri contratti. Controllano la validità delle transazioni a cui sono referenziati.
Tendermint	L'aggiornamento dello stato viene gestito dall'applicazione, formato coppia chiave-valore.	La business logic degli smart contract qui viene modellata dalle applicazioni ABCI.

Figura 4.5: Tabella riassuntiva del confronto della gestione dello stato e della declinazione del modello smart contract tra Hyperledger Fabric, Corda e Tendermint.

Capitolo 5

Conclusioni

In questa tesi si è stilato un modello generale dei sistemi blockchain, descrivendone gli elementi principali che li compongono e il funzionamento di base. Inoltre, sono state analizzate le varie problematiche tipiche dei sistemi distribuiti e di come la blockchain sia in grado di approcciarle e risolverle. In seguito, sono stati analizzati gli *smart contract* e si è cercato di dare loro una definizione, basando l'analisi su diverse interpretazioni al contesto date da diversi autori. Infine, sono state confrontate tra di loro alcune implementazioni di rilievo.

In seguito all'analisi svolta all'interno di questa tesi si può affermare che i sistemi basati su tecnologia blockchain sono sistemi molto versatili e il fatto che vengano sfruttati principalmente come base per lo sviluppo di criptovalute riduce l'impatto tecnologico che effettivamente potrebbero avere.

La tecnologia "blockchain" è riuscita a risolvere, con degli approcci particolari, alcuni dei problemi riguardanti i sistemi distribuiti, come la mancanza di clock globale all'interno del sistema, e a garantire la sicurezza dello stesso attraverso algoritmi di consenso distribuito. Gli **smart contract** sono, dal punto di vista concettuale, un ottimo punto di partenza per creare un'infrastruttura che apra le frontiere per lo sviluppo futuro di framework che daranno la possibilità di interfacciare questa nuova tecnologia verso molti più contesti, finanziari e non, tra cui l'**Internet Of Things**, l'**health-care** e l'**identity management**.

Questa tesi si è presa il compito di generalizzare il modello della tecnologia per dare a tutti la possibilità di comprendere il funzionamento del "motore" che permette ai sistemi di svolgere in maniera efficiente e sicura tutte le operazioni. L'analisi dettagliata dei problemi affrontati dalla blockchain, poi, dà al lettore una serie di spunti di riflessione sul come questa tecnologia possa essere ritenuta sicura sotto certi aspetti. Naturalmente, come ogni tecnologia, ha i suoi difetti, come dimostrano i documenti sugli attacchi e le varie problematiche che le varie blockchain hanno dovuto affrontare durante questi ultimi anni. Nonostante ciò, questo sistema si dimostra versatile, solido e ricco di potenziale. Potenziale che viene già espresso, seppur in maniera embrionale, dalle già citate *Filecoin* [10] e *Blockstack* [2], che avvalorano quanto detto in precedenza, ossia l'impiego di questa tecnologia al di fuori dei contesti finanziari e industriali.

Le tecnologie che sono state analizzate nel Capitolo 4 sono un ottimo punto di partenza per eventuali sviluppi futuri dell'argomento, con analisi più approfondite per quanto riguarda l'ambiente di sviluppo e le potenzialità delle varie implementazioni, toccate solo in parte dall'analisi proposta da questa tesi. Sarebbe opportuno approfondire, inoltre, i campi esplorati da *Filecoin* e *Blockstack*, vista l'innovazione tecnologica che vogliono mettere in gioco. *Filecoin* vuole creare uno storage cloud decentralizzato e libero, ossia un servizio dove le entità centralizzate che controllano gran parte dei cloud services, come Amazon, Google e Dropbox, non abbiano la possibilità di sapere esattamente quali siano i dati che l'utente va a salvare nel cloud, perché questi ultimi vengono criptati e solo chi fa parte del sistema messo a disposizione da *Filecoin* può usufruire di quei dati. Per quanto riguarda, invece, *Blockstack*, si prende carico del compito di cercare di decentralizzare la rete internet. *Blockstack* è essenzialmente composto da tre livelli: al livello più basso è presente una blockchain, che ha il compito di fornire un servizio di risoluzione dei nomi, una sorta di **DNS**, il secondo livello è composto da una rete peer-to-peer che funge da indice globale per la ricerca di informazioni e il livello più alto corrisponde a una serie di server dove sono memorizzati i vari dati.

In futuro, sarebbe utile affrontare un'analisi completa e dettagliata sul mondo degli algoritmi di consenso, in special modo gli algoritmi di consenso bizantino (algoritmi **BFT**), per poi stilarne un confronto ed analizzare l'aspetto dell'adozione di uno specifico algoritmo sulle applicazioni finali e le loro performance.

La tecnologia delle blockchain è ancora in uno stato immaturo, nonostante gli enormi progressi fatti negli ultimi anni. Il fattore che sta facendo conoscere questa tecnologia anche a chi non si occupa prettamente di informatica è il "fattore criptovalute". Essendo delle valute a tutti gli effetti, sono suscettibili al proprio valore di mercato (o al valore della più importante criptovaluta del momento, il **Bitcoin**, con un valore di mercato di quasi 9.000 €). Se il mercato delle criptovalute crollasse, l'interesse che il mondo ha verso questa tecnologia potrebbe via via scemare, fino a farla cadere in disuso, rischiando di sprecare le sue numerose potenzialità ancora inespresse. Questo perché i *miner* non troverebbero interesse a partecipare attivamente alla vita di una blockchain dato che non ci sarebbe per loro un ritorno economico adeguato. Per questo è necessario che la tecnologia "blockchain" venga presentata come un qualcosa che può essere applicata a molti più ambiti oltre a quello finanziario. Occorre quindi che la comunità che ruota intorno allo sviluppo di questa tecnologia inizi a proporla ad ambienti sensibili, come ospedali, e creare un sistema che possa mettere in contatto tra di loro istituzioni diverse sparse per il globo, in modo tale da creare una rete dove le informazioni vengono scambiate in maniera decentralizzata e libera. Per questi motivi, chi scrive ritiene che l'utilizzo della tecnologia "blockchain", o meglio, l'utilizzo di un sistema decentralizzato che gestisca una parte sostanziosa della vita quotidiana, come la rete Internet, sia ormai una cosa quasi augurabile visto che si sta lottando per avere una rete Internet libera da manipolazioni da parte dei governi, viste anche le recenti vicissitudini riguardanti la *net neutrality* negli Stati Uniti [32].

Ringraziamenti

Ringrazio innanzitutto il Prof. Andrea Omicini che mi ha dato la possibilità di sviluppare questa tesi su un argomento che mi ha affascinato sin dal momento in cui ne ho sentito parlare per la prima volta. Un ringraziamento sentito va anche al mio co-relatore, il Dott. Giovanni Ciatto, che mi ha aiutato molto durante questo periodo di tesi e di conoscenza del contesto. Il mio ringraziamento più sentito va ai miei genitori, che mi hanno permesso di studiare e non mi hanno fatto mai mancare il loro supporto morale. Supporto morale che mi è stato anche dato dai miei amici più cari. Un grazie a Lorenzo, Marco, Enrico, Davide Foschi, Daci, Sarah, Matteo, Lozzi, Ilaria, Filippo, Andrea, Cristian, Brando, Dataico e Giulia. Senza di loro avrei sicuramente perso molta più sanità mentale del dovuto. Un grazie anche a parte dello Staff e agli amici della chat di Italiansubs.net che mi hanno fatto svagare durante questo periodo di tesi.

Bibliografia

[1] K. Christidis and M. Devetsikiotis, "Blockchains and smart contracts for the internet of things," *IEEE Access*, vol. 4, pp. 2292–2303, 2016.

[2] M. Ali, R. Shea, J. Nelson, and M. J. Freedman, "Blockstack whitepaper." `https://blockstack.org/whitepaper.pdf`, 2017.

[3] Wikipedia, "Funzione di densità di probabilità." `https://it.wikipedia.org/wiki/Funzione_di_densit%C3%A0_di_probabilit%C3%A0`, 2017.

[4] Treccani, "Asset." `http://www.treccani.it/enciclopedia/asset_%28Dizionario-di-Economia-e-Finanza%29/`, 2017.

[5] A. S. Tanenbaum and M. v. Steen, *Distributed Systems: Principles and Paradigms (2Nd Edition)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2006.

[6] C. Cachin, E. Androulaki, A. D. Caro, M. Osborne, S. Schubert, A. Sorniotti, M. Vukolic, and T. Weigold, "Blockchain, cryptography, and consensus." `https://cachin.com/cc/talks/20161004-blockchain-techtuesday-web.pdf`, 2016.

[7] M. Vukolić, "The quest for scalable blockchain fabric: Proof-of-work vs. bft replication," pp. 112–125, 2015.

[8] Bitcoin, "Global bitcoin nodes distribution." `https://bitnodes.earn.com/`, 2018.

[9] A. Salomaa, *Public-Key Cryptography*. Springer Publishing Company, Incorporated, 2nd ed., 2010.

[10] P. Labs, "Filecoin whitepaper." https://filecoin.io/filecoin.pdf, 2017.

[11] S. Haber and W. S. Stornetta, "How to time-stamp a digital document," *Journal of Cryptology*, vol. 3, pp. 99–111, 1991.

[12] A. Narayanan, J. Bonneau, E. W. Felten, A. Miller, and S. Goldfeder, *Bitcoin and Cryptocurrency Technologies - A Comprehensive Introduction.* Princeton University Press, 2016.

[13] Rosargia, "Blockchain & cryptocurrency #2: Hash pointers and data structures." https://steemit.com/hash-pointer/@rosargia/hash-pointers-and-data-structures, 2017.

[14] J. J. Douceur, "The sybil attack," in *Proceedings of 1st International Workshop on Peer-to-Peer Systems (IPTPS)*, January 2002.

[15] Bitcoin, "Weaknesses, the attacker has a lot of computing power." https://en.bitcoin.it/wiki/Weaknesses#Attacker_has_a_lot_of_computing_power, 2018.

[16] M. J. A. Sexton, "The ethereum effect: Graphics card price watch." http://www.tomshardware.com/news/ethereum-effect-graphics-card-prices,34928.html, 2018.

[17] N. Szabo, "Smart contracts: Building blocks for digital markets." http://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart_contracts_2.html, 1996.

[18] J. Hopcroft, R. Motwani, J. Ullman, and G. Pighizzini, *Automi, linguaggi e calcolabilità.* Addison Wesley, Pearson, 2009.

[19] R. G. Brown, "A simple model for smart contracts." https://gendal.me/2015/02/10/a-simple-model-for-smart-contracts/, 2015.

[20] G. Greenspan, "Smart contracts: The good, the bad and the lazy." https://www.multichain.com/blog/2015/11/smart-contracts-good-bad-lazy/, 2015.

[21] I. Grigg, "The ricardian contract." `http://iang.org/papers/ricardian_contract.html`, 1990.

[22] F. B. Schneider, "Chapter 7: Replication management using the state-machine approach, distributed systems, 2nd edn," 1993.

[23] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Comput. Surv.*, vol. 22, pp. 299–319, Dec. 1990.

[24] Etherchain, "Dao attacker account." `https://www.etherchain.org/account/0x304a554a310c7e546dfe434669c62820b7d83490`, 2016.

[25] C. Metz, "The biggest crowdfunding project ever—the dao—is kind of a mess." `https://www.wired.com/2016/06/biggest-crowdfunding-project-ever-dao-mess/`, 2016.

[26] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on ethereum smart contracts (sok)," pp. 164–186, 03 2017.

[27] Hyperledger, "Hyperledger fabric documentation." `https://hyperledger-fabric.readthedocs.io/en/release/index.html`, 2017.

[28] Docker, "Docker." `https://www.docker.com/`, 2018.

[29] Hyperledger, "Hyperledger fabric explainer." `https://www.youtube.com/watch?v=js3ZjxbO8TM`, 2017.

[30] R3, "Corda." `https://www.corda.net/`, 2017.

[31] Cosmos, "Tendermint." `https://tendermint.com/`, 2017.

[32] G. Mosca, "Stati uniti, addio net neutrality." `https://www.wired.it/attualita/tech/2017/12/14/stati-uniti-addio-net-neutrality/`, 2017.